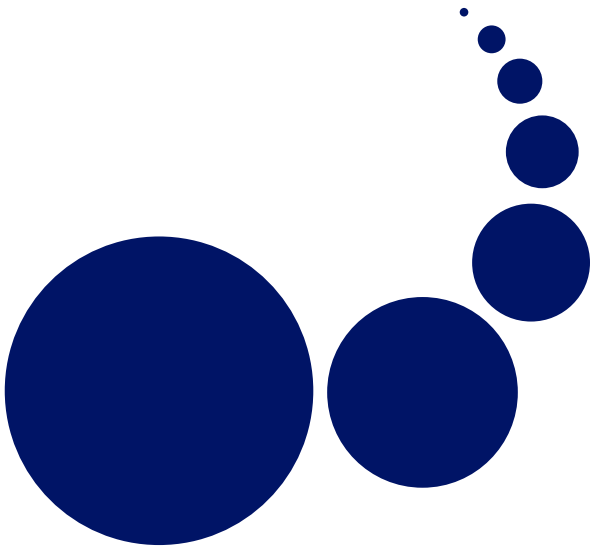
SCALABLE COMPUTING
Practice and Experience

Special Issue: Internet-based Computing

Editors: Dan Grigoras, John P. Morrison, Marcin Paprzycki



Volume 6, Number 1, March 2005

ISSN 1895-1767

Marcin Paprzycki

Institute of Computer Science
Warsaw School of Social Psychology
ul. Chodakowska 19/31
03-815 Warszawa
Poland
marcin.paprzycki@swps.edu.pl
http://mpaprzycki.swps.edu.pl

Institute of Applied Informatics
Wrocław University of Technology
Wyb. Wyspiańskiego 27
Wrocław 51-370, POLAND
pawel.myszkowski@pwr.wroc.pl

Graduate School
of Information Science,
Japan Advanced Institute
of Science & Technology
1-1 Asahidai, Tatsunokuchi,
Ishikawa 923-1292, JAPAN
shen@jaist.ac.ip

ISI-CNR c/o DEIS
Università della Calabria
87036 Rende, CS, ITALY
talia@si.deis.unical.it

Elbląg University
of Humanities and Economy
ul. Lotnicza 2
82-300 Elbląg, POLAND
denisjuk@euh-e.edu.pl

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 6, Number 1, March 2005

TABLE OF CONTENTS

ISSN 1895-1767

INSTEAD OF THE EDITORIAL

Dear Readers,

This time, instead of the usual Editorial written by one of my colleagues, I am writing a brief explanatory note. As many of you have noticed, over the last 3 years, the speed of publication of our PDCP journal has slowed beyond any reasonable pace. I do not want to point fingers at the publisher as in some ways we share the blame. Instead I prefer to look into the future. After the initial 5 years contract has run out, we have decided that it will be in the best interest if we (Editors and our Publisher) parted our ways.

Since we were to go through changes anyway, we have decided to make a number of changes. First, in an effort to acknowledge the changes that occurred in the world of computing, our journal will change its name to *Scalable Computing: Practice and Experience.* Second, since the new journal will be published by Warsaw School of Social Psychology (`http://www.swps.edu.pl`) it will receive a new ISSN number. Furthermore, to recognize the fact that the new journal is a continuation of the original *PDCP* we have decided to continue the numbering structure and thus the first issue of the new journal will be published as Volume 6, Number 1. However, to make clear that the new journal started appearing in 2005, this will be the date of publication.

The most important decision, however, was to make the new journal appear only in an electronic form. Let me stress, that while each contribution will be refereed by at least two referees and we will publish only very high quality submissions, the form of publication will be **electronic only.** For each issue we will publish a single PDF/PS file containing a complete issue and separate files containing each paper.

Since Warsaw School of Social Psychology became a publisher and sponsor of our journal, we will be able to provide content free of charge. Furthermore, we would like to offer a subscription service that will allow you to receive tables of content of published issues. Subscription mechanism requires address confirmation and we promise that personal information stored in our database will remain completely confidential and will not be sold and/or shared with anyone.

Our first order of business is to clean the queue of old special issues and papers. And we plan to complete this process by the end of this year. In the meantime I would like to invite you to submit papers and/or suggest Special Issues. Let us make the "new SCPE" a great e-journal.

Sincerely Yours,
Marcin Paprzycki
Editor-in-Chief


SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 6, Number 1, pp. iii. http://www.scpe.org

INTRODUCTION TO THE SPECIAL ISSUE: INTERNET-BASED COMPUTING

Dear Readers,

It is our pleasure to present to you a long-awaited special issue of the PDCP journal. Since the issue is finally published in the rejuvenated PDCP, we will not delve into the reasons for the delay. Instead, we present to you 8 extended papers that were originally presented during the First International Symposium on Parallel and Distributed Computing that took place in Iasi, Romania, in August, 2001. Out of the collection of very good papers we have selected these that dealt with Internet-oriented computing. The first three pages deal with peer-to-peer and meta computing: W. Hoschek "Enabling Rich Service and Resource Discovery with Database for Dynamic Distributed Content", K. Power, J. P. Morrison "Ad Hoc Metacomputig With Compeer" and J. P. Morrison, P. D. Healy "Implementing a Condensed Graphs Based Metacomputer with XML". The next three papers deal with various issues in performance analysis: G. Da Costa, R. Olivier "Impact of Realistic Workload in Peer-to-Peer Systems a Case Study—Freenet", E. Caron, F. Desprez, F. Suter "Parallel Extension of a Dynamic Performance Forecasting Tool" and R. Harakaly, P. Primet, F. Bonnassieux, B. Gaidioz "Probes Coordination Protocol for Network Performance Measurement in GRID Computing Environment". The remaining three papers are: D. Weyns, E. Eddy, P. Verbaeten "Serialization of Distributed Threads in Java," V. Fiolet, B. Toursel "Distributed Data Mining," and W. Gawinecki, M. Gordon, P. Kaczmarek, M. Paprzycki "The Problem of Agent-Client Communication on the Internet".

We believe that even though time has passed since original preparation, the quality of the papers remains high and they are definitely worthy your attention.

Dan Grigoras
John Morrison
Marcin Paprzycki

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 1–16. http://www.scpe.org
ISSN 1895-1767

ENABLING RICH SERVICE AND RESOURCE DISCOVERY WITH A DATABASE FOR DYNAMIC DISTRIBUTED CONTENT

WOLFGANG HOSCHEK*

Abstract. In a distributed system such as a Data Grid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. However, it is not obvious how a database (registry) should maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. In particular, how can one avoid sacrificing reliability, predictability and simplicity while allowing to express powerful queries over time-sensitive dynamic information? We propose the so-called *hyper registry*, which has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

Key words. Dynamic Database, XQuery, Service Discovery

1. Introduction. The next generation Large Hadron Collider project at CERN, the European Organization for Nuclear Research, involves thousands of researchers and hundreds of institutions spread around the globe. A massive set of computing resources is necessary to support it's data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [1]. To make collaboration viable, it was decided to share in a global joint effort—the European Data Grid (EDG) [2, 3]—the data and locally available resources of all participating laboratories and university departments.

Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [4]. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

An enabling step towards increased Grid software execution flexibility is the (still immature and hence often hyped) *web services* vision [5, 6, 7] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible, adaptive and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. For example, a data-intensive High Energy Physics analysis application sweeping over Terabytes of data looks for remote services that exhibit a suitable combination of characteristics, including network load, available disk quota, access rights, and perhaps Quality of Service and monetary cost. It is thus of critical importance to develop capabilities for rich service discovery as well as a query language that can support advanced resource brokering.

More generally, in a distributed system, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. For example, it allows a search for descriptions of services of a file sharing system, to determine its total download capacity, the names of all participating organizations, etc. Example systems include a service discovery system for a (worldwide) Data Grid, an electronic market place, or an instant messaging and news service. In all these systems, a variety of information describes the state of autonomous remote participants residing within different administrative domains. Participants frequently join, leave and act on a best effort basis. In a large distributed system spanning many administrative domains, predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception.

*CERN IT Division, European Organization for Nuclear Research, 1211 Geneva 23, Switzerland(wolfgang.hoschek@cern.ch).

In this paper, a design and specification for a type of database is developed that is conditioned to address the problem, the so-called *hyper registry*. A hyper registry has a database that holds a set of tuples. A *tuple* may contain an arbitrary piece of arbitrary *content*. Examples of content include a service description expressed in WSDL [8], a Quality of Service description, a file, file replica location, current network load, host information, stock quotes, etc., as depicted in Figure 1.1. A tuple is annotated with a *content link* pointing to the authoritative data source of the embedded content.

```
<service>
  <interface type = "http://gridforum.org/interface/scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/scheduler/submitjob"/>
    </operation>
  </interface>
</service>
```

```
<hostInfo>
   <host name="fred01.cern.ch" os="i386_redhat 7.2"   MHz="1000" cpus="2"/>
   <host name="fred02.cern.ch" os="sparc_solaris 2.7" MHz="400"  cpus="64"/>
</hostInfo>
```

```
<replicaSet LFN="urn:/iana/dns/ch/cern/cms/higgs" size="1000000" type="MySQL">
   <PFN URL="ftp://storage.cern.ch/file123" readCount="17"/>
   <PFN URL="ftp://se01.infn.it/file456"    readCount="1"/>
</replicaSet>
```

FIG. 1.1. *Example Content.*

This paper is organized as follows. Section 2 identifies query types and requirements by means of examples. Section 3 discusses content description and publication. A *content provider* can publish a dynamic pointer called a *content link*, which in turn enables the hyper registry and third parties to retrieve (pull) the current content presented from the provider at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). Section 4 illustrates how a remote client can query a hyper registry, obtaining a set of tuples as answer. The use of XQuery [9, 10] as a rich and expressive query language is motivated and discussed by means of examples. Section 5 proposes solutions to the cache coherency issues arising from content *caching* for improved client efficiency. For reliable, predictable and simple state maintenance, we propose to keep a registry tuple as soft state. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from the content provider. Section 6 shows that content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. Section 7 compares our approach with related work. Finally, Section 8 summarizes and concludes this paper.

2. Query Examples and Types. To concretize discussion and to identify query types and requirements, we now give several example queries related to service discovery. Queries are initially expressed in prose. Some of them will later be formalized in a suitable query language. One can distinguish three types of queries: *simple, medium* and *complex*. The latter are more powerful than the former. Nevertheless, even a simple query is a powerful tool.

2.1. Simple Query. Simple queries are most often used for discovery. A simple query finds all tuples (services) matching a given predicate or pattern. The query visits each tuple (service description) in a set individually, and generates a result set by applying a function to each tuple. The function usually consists of a predicate and/or a transformation. Individual answers are added to a (initially empty) result set. An empty answer leaves the result set unchanged. A simple query has the following form:

```
R = {}
for each tuple in input
```

```
  R = R UNION { function(tuple) }
endfor
return R
```

Example simple queries are:

- *(QS1) Find all (available) services.*
- *(QS2) Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN)".*
- *(QS4) Find all local services (all service interfaces of any given service must reside on the same host).*
- *(QS5) Find all services and return their service links (instead of descriptions).*
- *(QS6) Find all CMS replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*
- *(QS7) Within the domain "cern.ch", find all execution services and their CPU load where cpuLoad* < 0.5 *(Assuming the operation cpuLoad() is defined on the execution service).*

In support of the wide variety of real-life questions anticipated, it should be possible to arbitrarily combine and nest all capabilities exposed in these examples. Note that the first four queries return service descriptions, whereas the others return additional or entirely different information (service links, physical file names or CPU load). We term the former queries *predicate (or filter) queries*. The structure of the result set is predetermined in the sense that query output must be a subset of query input. We term the latter queries *constructive queries*, because they construct answers of arbitrary structure and content. Predicate queries are a subset of constructive queries. A constructive query function that always returns `"Hello World"` or an empty string is legal, but not very useful.

Further, note that the queries *QS6, QS7* involve multiple independent data sources and match on dynamically delivered content (via remote invocation of operations `getPFNs` and `cpuLoad`), rather than on values being part of service descriptions. We call these queries *dynamic queries*, as opposed to *static queries*. To support dynamic queries, a query language must provide means to dynamically retrieve and interpret information from diverse remote or local sources.

Dynamic queries can sometimes be reformulated as static queries. For example, the LFN/PFN database information of query *QS6* could be made available as part of the tuple set. In practice, this is typically infeasible for reasons including database size, consistency, information hiding, security and performance. Publishing highly volatile attributes such as CPU load as part of tuples leads to stale data problems. Clearly dynamic invocation is a more appropriate vehicle to deliver CPU load. Alternatively, custom push protocols can be used [11].

2.2. Medium query. A medium query computes an answer over a set of tuples (service descriptions) as a whole. For example, it can compute aggregates like number of tuples, maximum, etc. Examples of medium queries are:

- *(QM1) Find the CMS storage service with the largest network bandwidth to my host "dummy.cern.ch" (assuming there exists a service estimating bandwidth from A to B).*
- *(QM2) Return the number of replica catalog services.*
- *(QM3) Find the two CMS execution services with minimum and maximum CPU load and return their service descriptions and load.*
- *(QM4) Return a summary of all replica catalogs and schedulers residing within the domains "cern.ch", "infn.it" and "anl.gov", grouped in ascending order by owner, domain and service type, with aggregate group cardinalities.*

The query is applied to the set as a whole. For example, *QM4* is interesting in that it involves crossing tuple boundaries, which simple hierarchical query languages typically do not support. Like a simple query, a medium query can be static or dynamic. It can be a predicate query or a constructive query.

2.3. Complex query. Complex queries are most often used for advanced discovery or brokering. Like a medium query, a complex query computes an answer over a set of tuples (service descriptions) as a whole. However, it has powerful capabilities to combine data from multiple sources. For example, it supports all database join flavors. Like any other query, a complex query can be static or dynamic. It can be a predicate query or a constructive query. Example complex queries are:

- *(QC1) Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*
- *(QC2) Find all domains that run more than one replica catalog with CMS as owner. (Want to check for anomalies).*
- *(QC3) Find the top 10 owners of replica catalog services within the domains "cern.ch", "infn.it" and "anl.gov", and return their email, together with the number of services each of them owns, sorted by that number.*

3. Content Link, Content Provider and Publication. A *content provider* can publish a dynamic pointer called a *content link*, which in turn enables the hyper registry and third parties to retrieve (pull) the current content presented from the provider at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push).

3.1. Content Link. A *content link* may be any arbitrary URI. However, most commonly, it is an HTTP(S) URL, in which case it points to the content of a content provider, and an HTTP(S) GET request to the link must return the current (up-to-date) content. In other words, a simple hyperlink is employed. In the context of service discovery, we use the term *service link* to denote a content link that points to a service description. Content links can freely be chosen as long as they conform to the URI and HTTP URL specification [12]. Examples of content links are:

```
urn:/iana/dns/ch/cern/cn/techdoc/94/1642-3
urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6
http://sched.cern.ch:8080/getServiceDescription.wsdl
https://cms.cern.ch/getServiceDesc?id=4712&cache=disable
http://phone.cern.ch/lookup?query="select phone from book where phone=4711"
http://repcat.cern.ch/getPFNs?lfn="myLogicalFileName"
```

3.2. Content Provider. A *content provider* offers information conforming to a homogeneous global data model. In order to do so, it typically uses some kind of internal mediator to transform information from a local or proprietary data model to the global data model. A content provider can be seen as a gateway to heterogeneous content sources. The global data model is the Dynamic Data Model (DDM) introduced in Section 3.3. Content can be structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. Individual content may, but need not, have a schema (XML Schema [13]), in which case content must be valid according to the schema. All content may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content.

A content provider is an umbrella term for two components, namely a presenter and a publisher. The *presenter* is a service and answers HTTP(S) GET content retrieval requests from a hyper registry or client (subject to local security policy). The *publisher* is a piece of code that publishes content link, and perhaps also content, to a hyper registry. The publisher need not be a service, although it uses HTTP(S) POST for transport of communications. The structure of a content provider and its interaction with a hyper registry and a client are depicted in Figure 3.1 (a). Note that a client can bypass a hyper registry and directly pull current content from a provider. Figure 3.1 (b) illustrates a hyper registry with several content providers and clients.

Just as in the dynamic WWW that allows for a broad variety of implementations for the given protocol, it is left unspecified how a presenter computes content on retrieval. Content can be static or dynamic (generated on the fly). For example, a presenter may serve the content directly from a file or database, or from a potentially outdated cache. For increased accuracy, it may also dynamically recompute the content on each request. Consider the example providers in Figure 3.2. A simple but nonetheless very useful content provider uses a commodity HTTP server such as Apache to present XML content from the file system. A simple `cron` job monitors the health of the Apache server and publishes the current state to a hyper registry. Another example for a content provider is a Java servlet that makes available data kept in a relational or LDAP database system. A content provider can execute legacy command line tools to publish system state information such as network statistics, operating system and type of CPU. Another example for a content provider is a network service such as a replica catalog that (in addition to servicing replica lookup requests) publishes its service description and/or link so that clients may discover and subsequently invoke it.

Providers and registries can be deployed and configured arbitrarily. For example, in a strategy for scalable administration of large cluster environments, a single shared Apache web server can easily be configured to

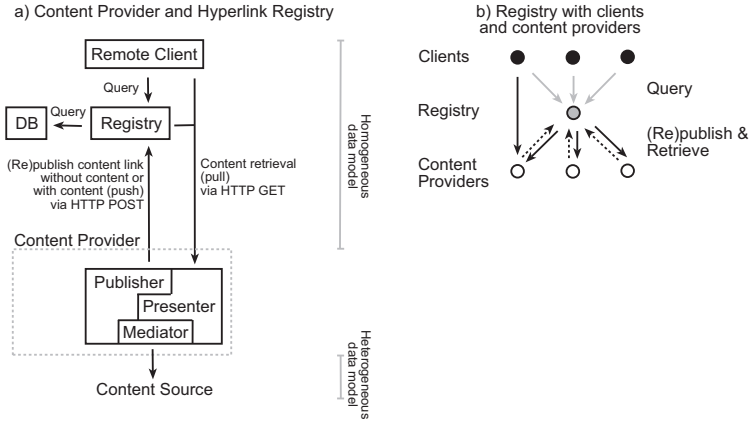


FIG. 3.1. *Content Provider and Hyper Registry.*

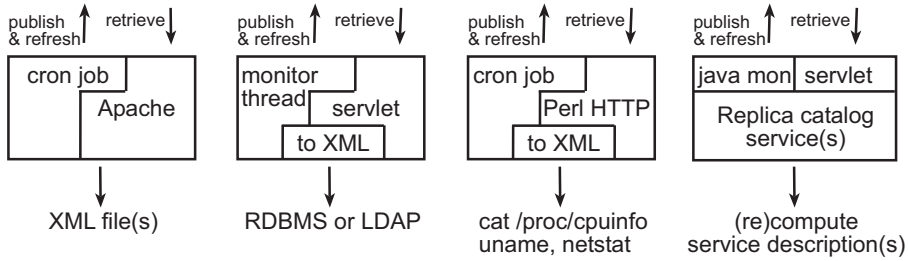


FIG. 3.2. *Example Content Providers.*

serve XML descriptions of thousands of services on hundreds of hosts. For example, via a naming convention we can assign a distinct web server directory and corresponding service link for each host-service combination. To serve descriptions it is sufficient to have some administrative `cron` job run periodically on each service host, which writes the current service description into an appropriate XML file in the appropriate directory on the web server.

Presenter and publisher are not required to run in the same process or even on the same host. For efficiency, a so-called *container* of a virtual hosting environment may be used to run more than one provider in the same process or thread. For example, a highly efficient and scalable container such as the Apache Tomcat servlet engine [14] not only can serve many hundreds to a thousand (light-weight) concurrent requests per second on a commodity PC, but it can also embed any number of dynamic provider types in the same process, with each provider type being capable of serving any number of provider instances. Typically, a provider is remote to the hyper registry. This is not a requirement, however. A provider may also be local to the hyper registry and connect through a loop-back connection. For further efficiency, a hyper registry may internally host any number of built in providers. Using commodity servlet technology, any number of hyper registries can be hosted within the same container.

3.3. Publication. In a given context, a content provider can publish content of a given type to one or more registries. More precisely, a content provider can publish a dynamic pointer called a content link, which in turn enables the hyper registry (and third parties) to retrieve the current content. For efficiency, the `publish` operation takes as input a set of zero or more tuples. In what we propose to call the *Dynamic Data Model (DDM)*, each XML tuple has a content link, a type, a context, some soft state time stamps, and (optionally) metadata and content. A tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary *content* and allows for refresh of that content at any time. Consider the dynamic tuple set depicted in Figure 3.3.

- **Link.** The content link is an HTTP(S) URL as introduced above. Given the link the current content of a content provider can be retrieved (pulled) at any time.
- **Type.** The type describes *what* kind of content is being published (e.g. `service`,

```
<tupleset>
    <tuple link="http://registry.cern.ch/getDescription" type="service" ctx="parent"
           TS1="10" TC="15" TS2="20" TS3="30">
       <content>
         <service>
           <interface type="http://cern.ch/Presenter-1.0">
             <operation>
               <name>XML getServiceDescription()</name>
               <bind:http verb="GET" URL="https://registry.cern.ch/getDesc"/>
             </operation>
           </interface>
           <interface type = "http://cern.ch/XQuery-1.0">
             <operation>
               <name> XML query(XQuery query)</name>
               <bind:beep URL="beep://registry.cern.ch:9000"/>
             </operation>
           </interface>
         </service>
       </content>
       <metadata>  <owner name="http://cms.cern.ch"/> </metadata>
    </tuple>
    <tuple link="http://repcat.cern.ch/getDesc?id=4711" type="service" ctx="child"
           TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>
    <tuple link="urn:uuid:f81d4fae-11d0-a765-00a0c91e6bf6"
           type="replica" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <replicaSet LFN="urn:/iana/dns/ch/cern/cms/higgs" size="1000000" type="MySQL">
             <PFN URL="ftp://storage.cern.ch/file123" readCount="17"/>
             <PFN URL="ftp://se01.infn.it/file456"    readCount="1"/>
         </replicaSet>
       </content>
    </tuple>
    <tuple link="http://hosts.cern.ch" type="hosts" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <hosts>
             <host name="fred1.cern.ch" os="rh7.2"  arch="i386"  mem="512M" MHz="1000"/>
             <host name="fred2.cern.ch" os="sol2.7" arch="sparc" mem="8192M" MHz="400"/>
         </hosts>
       </content>
    </tuple>
</tupleset>
```

FIG. 3.3. *Example Tuple Set from Dynamic Data Model.*

`application/octet-stream`, `image/jpeg`, `networkLoad`, `hostinfo`).

- **Context.** The context describes *why* the content is being published or *how* it should be used (e.g. `child, parent, x-ireferral, gnutella, monitoring`). Note that it is often not meaningful to embed context information in the content itself, because a given content can be published in a different context at different hyper registries (1:N association). For example, a service can be a child of some nodes and at the same time be a parent for some other nodes. However, its service description (content) should clearly remain invariant. In addition, context and type allow a query to differente on crucial attributes even if content caching is not supported or not authorized.
- **Timestamps TS1, TS2, TS3, TC.** Discussion of timestamps is deferred to Section 5.2 below.
- **Metadata.** The optional metadata element further describes the content and/or its retrieval beyond what can be expressed with the previous attributes. For example, the metadata may be a secure digital XML signature [15] of the content. It may describe the authoritative content provider or owner of

the content. Another metadata example is a Web Service Inspection Language (WSIL) document [16] or fragment thereof, specifying additional content retrieval mechanisms beyond HTTP content link retrieval. The metadata argument may be any well-formed XML document or fragment. It is an extensibility element enabling customization and flexible evolution.

- **Content.** Given the link the current content of a content provider can be retrieved (pulled) at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). For clarity of exposition, the published content is an XML element[1].

The publish operation of a hyper registry has the signature `void publish(XML tupleset)`. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. If a key does not already exist on publication, a tuple is inserted into the hyper registry database. An existing tuple can be updated by publishing other values under the same tuple key. An existing tuple (key) is "owned" by the content provider that created it with the first publication. It is recommended that a content provider with another identity may not be permitted to publish or update the tuple.

- *Find all (available) services.*

```
RETURN /tupleset/tuple[@type="service"]
```

- *Find all services that implement a replica catalog service interface that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN)".*

```
LET $repcat := "http://gridforum.org/interface/replicaCatalog-1.0"
FOR $tuple IN /tupleset/tuple[@type="service"]
WHERE SOME $op IN $tuple/content/service/interface[@type = $repcat]/operation
      SATISFIES ($op/name="XML getPFNs(String LFN)" AND $op/bindhttp/@verb="GET"
                 AND contains($op/allow, "http://cms.cern.ch/everybody"))
RETURN $tuple
```

- *Return the number of replica catalog services.*

```
LET $repcat := "http://gridforum.org/interface/replicaCatalog-1.0"
RETURN count(/tupleset/tuple/content/service[interface/@type=$repcat])
```

- *Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

```
LET $exeType := "http://gridforum.org/interface/executor-1.0"
LET $stoType := "http://gridforum.org/interface/storage-1.0"
FOR $executor IN/tupleset/tuple[content/service/interface/@type = $exeType],
    $storage  IN /tupleset/tuple[content/service/interface/@type = $stoType AND
    domainName(@link) = domainName($executor/@link)]
RETURN <pair> {$executor} {$storage} </pair>
```

FIG. 3.4. *Simple, Medium and Complex XQueries for Service Discovery.*

4. Query.

4.1. Minimalist Query. Clients can query the hyper registry by invoking minimalist query operations (*"select all"*-style). The `getTuples()` query operation takes no arguments and returns the full set of all tuples "as is". That is, query output format and publication input format are the same (see Figure 3.3). If supported, output includes cached content. The `getLinks()` query operation is similar in that it also takes no arguments and returns the full set of all tuples. However, it always substitutes an empty string for cached content. In other words, the content is omitted from tuples, potentially saving substantial bandwidth. The second tuple in Figure 3.3 has such a form.

4.2. XQuery. A hyper registry node has the capability to execute XQueries over the set of tuples it holds in its database. XQuery [9, 10] is the standard XML query language developed under the auspices of the

[1]In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME object. The XML based publication input set and query result set is augmented with an additional MIME multipart object [17], which is a list containing all content. The content element of the result set is interpreted as an index into the MIME multipart object. A typical hyper registry that supports caching can handle content with at least MIME content-type `text/xml` and `text/plain`.

W3C. It allows for powerful searching, which is critical for non-trivial applications. Everything that can be expressed with SQL [18] can also be expressed with XQuery. However, XQuery is a more expressive language than SQL. Simple, medium and complex XQueries for service discovery are depicted in Figure 3.4. XQuery can dynamically integrate external data sources via the `document(URL)` function, which can be used to process the XML results of remote operations invoked over HTTP. For example, given a service description with a `getNetworkLoad()` operation, a query can match on values dynamically produced by that operation. For a detailed discussion of a wide range of discovery queries, their representation in the XQuery language, as well as detailed motivation and justification, see our prior studies [5]. The same rules that apply to minimalist queries also apply to XQuery support. An implementation can use a modular and simple XQuery processor such as `Quip` [19] for the operation `XML query(XQuery query)`. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

4.3. Deployment. For flexibility, a hyper registry may be deployed in any arbitrary way (*deployment model*). For example, the database can be kept in a XML file, in the same format as returned by the `getTuples` query operation. However, tuples can certainly also be dynamically recomputed or kept in a remote relational database (table), for example as follows:

Link	Context	Type	TS1	TC	TS2	TS3	Content
`http://sched001.cern.ch/ getServiceDescription`	Parent	Service	10	15	20	30	<service> A </service>
`http://sched.infn.it:8080/pub/ getServiceDescription`	Child	Service	20	25	30	40	<service> B </service>
`http://repcat.cern.ch/pub/ getServiceDescription?id=4711`	Child	Service	30	0	40	50	null
`http://repcat.cern.ch/pub/ getStatistics`	Null	RepStats	60	65	70	80	<repcatStats> ... </repcatStats>

5. Caching and Soft State.

5.1. Caching. Content *caching* is important for client efficiency. The hyper registry may not only keep content links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A hyper registry may (but need not) support caching. A hyper registry that does not support caching ignores any content handed from a content provider. It keeps content links only. Instead of cached content it returns empty strings. Cache coherency issues arise. The query operations of a caching hyper registry may return tuples with stale content, i.e. content that is out of date with respect to its master copy at the content provider.

A caching hyper registry may implement a *strong* or *weak cache coherency policy*. A strong cache coherency policy is *server invalidation* [20]. Here a content provider notifies the hyper registry with a publication tuple whenever it has locally modified the content. We use this approach in an adapted version where a caching hyper registry can operate according to the client push pattern (*push hyper registry*) or server pull pattern (*pull hyper registry*) or a hybrid thereof. The respective interactions are as follows:

- **Pull Hyper Registry.** A content provider publishes a content link. The hyper registry then pulls the current content via content link retrieval into the cache. Whenever the content provider modifies the content, it notifies the hyper registry with a publication tuple carrying the time the content was last modified. The hyper registry may then decide to pull the current content again, in order to update the cache. It is up to the hyper registry to decide if and when to pull content. A hyper registry may pull content at any time. For example, it may dynamically pull fresh content for tuples affected by a query. This is important for frequently changing dynamic data such as network load.
- **Push Hyper Registry.** A publication tuple pushed from a content provider to the hyper registry contains not only a content link but also its current content. Whenever a content provider modifies content, it pushes the new content to the hyper registry, which may update the cache accordingly.

- **Hybrid Hyper Registry.** A hybrid hyper registry implements both pull and push interactions. If a content provider merely notifies that its content has changed, the hyper registry may choose to pull the current content into the cache. If a content provider pushes content, the cache may be updated with the pushed content. This is the type of hyper registry subsequently assumed whenever a caching hyper registry is discussed.

A non-caching hyper registry ignores content elements, if present. A publication is said to be *without content* if the content is not provided at all. Otherwise, it is said to be *with content*. Publication without content implies that no statement at all about cached content is being made (neutral). It does *not* imply that content should not be cached or invalidated.

A client must not assume that content is cached. For example, a hyper registry may not implement caching or it may be denied authorization when attempting to retrieve a given content, or it may ignore content provided on provider push. While it may be harmless that potentially anybody can learn that some content exists (content link), stringent trust delegation policies may dictate that only a few select clients, not including the hyper registry, are allowed to retrieve the content from a provider. Consider for example, that a detailed service description may be helpful for launching well-focused security attacks. In addition, a hyper registry may have an authorization policy. For example, depending on the identity of a client, a registry may return a subset of the full result set or hide cached content and instead return the tuple substituted by empty content. Similarly, depending on content provider identity, pushed content may be ignored or rejected, or publication denied altogether.

5.2. Soft State. For reliable, predictable and simple distributed state maintenance, a hyper registry tuple is maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from a content provider. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present for a further time.

The strong cache coherency policy *server invalidation* is extended. For flexibility and expressiveness, the ideas of the Grid Notification Framework [21] are adapted. The publication operation takes four absolute time stamps `TS1, TS2, TS3, TC` per tuple. The semantics are as follows. The content provider asserts that its content was last modified at time `TS1` and that its current content is expected to be valid from time `TS1` until at least time `TS2`. It is expected that the content link is alive between time `TS1` and at least time `TS3`. Time stamps must obey the constraint $\texttt{TS1} \leq \texttt{TS2} \leq \texttt{TS3}$. `TS2` triggers expiration of cached content, whereas `TS3` triggers expiration of content links. Usually, `TS1` equals the time of last modification or first publication, TS2 equals `TS1` plus some minutes or hours, and `TS3` equals `TS2` plus some hours or days. For example, `TS1`, `TS2` and `TS3` can reflect publication time, 10 minutes, and 2 hours, respectively.

A tuple also carries a timestamp `TC` that indicates the time when the tuple's embedded content (not the provider's master copy of the content) was last modified, typically by an intermediary in the path between client and content provider (e.g. the hyper registry). If a content provider publishes with content, then we usually have `TS1=TC`. `TC` must be zero-valued if the tuple contains no content. Hence, a hyper registry not supporting caching always has `TC` set to zero. For example, a highly dynamic network load provider may publish its link without content and `TS1=TS2` to suggest that it is inappropriate to cache its content. Constants are published with content and `TS2=TS3=infinity, TS1=TC=currentTime`.

These soft state time stamp semantics are summarized in Figure 5.1.

Time Stamp	**Semantics**
`TS1`	Time provider last modified content
`TC`	Time hyper registry last modified content cache
`TS2`	Expected time while current content at provider is at least valid
`TS3`	Expected time while content link at provider is at least valid (alive)

FIG. 5.1. *Soft State Time Stamp Semantics.*

Insert, update and delete of tuples occur at the timestamp-driven state transitions summarized in Figure 5.2.

Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. A tuple can be in one of three states: *unknown*, *not cached*, or *cached*. A tuple is unknown if it is not contained in the hyper registry (i.e. its key does not exist). Otherwise, it is known. When a tuple is assigned *not cached* state, its last internal modification time `TC` is (re)set to zero and the cache is deleted, if present. For a *not cached* tuple we have `TC < TS1`. When a tuple is assigned *cached* state, the content is updated and `TC` is set to the current time. For a *cached* tuple, we have `TC` $\geq$ `TS1`.

A tuple moves from *unknown* to *cached* or *not cached* state if the provider publishes with or without content, respectively. A tuple becomes *unknown* if its content link expires (`currentTime > TS3`); the tuple is then deleted. A provider can force tuple deletion by publishing with `currentTime > TS3`. A tuple is upgraded from *not cached* to *cached* state if a provider push publishes with content or if the hyper registry pulls the current content itself via retrieval. On content pull, a hyper registry may leave `TS2` unchanged, but it may also follow a policy that extends the lifetime of the tuple (or any other policy it sees fit). A tuple is degraded from *cached* to *not cached* state if the content expires. Such expiry occurs when no refresh is received in time (`currentTime > TS2`), or if a refresh indicates that the provider has modified the content (`TC < TS1`).

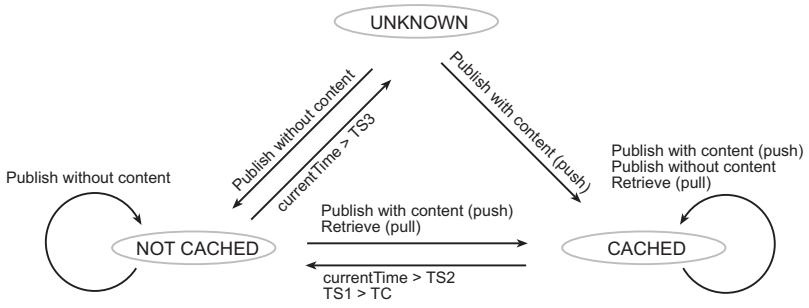


FIG. 5.2. *Soft State Transitions.*

6. Flexible Freshness. Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness. For example, a content provider can model the freshness of its content via pushing appropriate timestamps and content. A hyper registry can model the freshness of its content via controlled acceptance of provider publications and by actively pulling fresh content from the provider. If a result (e.g. network statistics) is up to date according to the hyper registry, but out of date according to the client, the client can pull fresh content from providers as it sees fit. However, this is inefficient for large result sets. Nevertheless, it is important for clients that query results are returned according to their notion of freshness, in particular in the presence of frequently changing dynamic content.

Recall that it is up to the hyper registry to decide to what extent its cache is stale, and if and when to pull fresh content. For example, a hyper registry may implement a policy that dynamically pulls fresh content for a tuple whenever a query touches (affects) the tuple. For example, if a query interprets the content link URL as an identifier within a hierarchical name space (e.g. as in LDAP) and selects only tuples within a sub-tree of the name space, only these tuples should be considered for refresh.

6.1. Refresh-on-client-demand. So far, a hyper registry must guess what a client's notion of freshness might be, while at the same time maintaining its decisive authority. A client still has no way to indicate (as opposed to force) its view of the matter to a hyper registry. We propose to address this problem with a simple and elegant *refresh-on-client-demand* strategy under control of the hyper registry's authority. The strategy exploits the rich expressiveness and dynamic data integration capabilities of the XQuery language. The client query may itself inspect the time stamp values of the set of tuples. It may then decide itself to what extent a tuple is considered interesting yet stale. If the query decides that a given tuple is stale (e.g. if `type="networkLoad" AND TC < currentTime() - 10`), it calls the XQuery `document(URL contentLink)` function with the corresponding content link in order to pull and get handed fresh content, which it then processes in any desired way.

This mechanism makes it unnecessary for a hyper registry to guess what a client's notion of freshness might be. It also implies that a hyper registry does not require complex logic for query parsing, analysis, splitting, merging, etc. Moreover, the fresh results pulled by a query can be reused for subsequent queries. Since the query is executed within the hyper registry, the hyper registry may implement the `document` function such that it not only pulls and returns the current content, but as a side effect also updates the tuple cache in its database. A hyper registry retains its authority in the sense that it may apply an authorization policy, or perhaps ignore the query's refresh calls altogether and return the old content instead. The refresh-on-client-demand strategy is simple, elegant and controlled. It improves efficiency by avoiding overly eager refreshes typically incurred by a guessing hyper registry policy.

This basic idea could be used in variations that are more ambitious. For example, in an attempt to improve efficiency via "batching", a query may collect the content links of all tuples it considers stale into a set, and hand the set to a `documents(URL[])` function provided by the hyper registry, which then fetches fresh content in a batched fashion. Alternatively, a query may use this approach in a non-blocking manner, merely indicating that the hyper registry should soon refresh the given tuples (asynchronously), while the old tuples are still fine for the current query. The theme can be developed further. In a hierarchical Peer-to-Peer environment with caching nodes along the query route, it may be preferable to have the `documents(URL[])` function forward the refresh set as a query to the next node along the query route, instead of directly pulling from the content provider. This refreshes all node caches along the route, possibly at the expense of increased latency. As a different type of optimization, the hyper registry may reduce latency by keeping alive the TCP connections to content providers, which is often impractical to do for clients.

To summarize, a wide range of dynamic content freshness policies can be supported, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness.

6.2. Throttling. Clearly there is a tradeoff between the resource consumption caused by refreshes and state consistency. The higher the refresh frequency, the more consistent and up-to-date the state, and the more resources are consumed. High frequency refresh can consume significant network bandwidth, due to pathological client misbehavior, denial-of-service attacks, or sheer popularity. Implementations using high frequency refresh rates can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) network connection setup for publication. Keep-alive connections should be used to minimize setup time. However, they only partly address the problem. Consider, for example, an automatically adapting search engine indexing one million services, each refreshing every minute with a message of 200 bytes. Just to stay up-to-date the search engine must scale to 17000 refreshes/sec and its maintainer must pay for a WAN bandwidth of at least 3.4 MB/sec.

To condition for overload, limit resource consumption and satisfy minimum requirements on content freshness, mechanisms to *throttle* refresh frequency are proposed, adaptively inviting more or less traffic over time. For example, the search engine hyper registry may ask the content providers to wait at least 100 minutes between refreshes. Conversely, a hyper registry of a job scheduler depending on very fresh CPU load measurements from job execution services may want to invite execution service providers to refresh at least every second. For example, the service administrator can set the aggregate bandwidth available for refresh to a fraction of the total available system bandwidth. [22, 23] suggest to determine the maximum refresh rate for a given source service from historic bandwidth statistics over refreshes. This avoids overload, yet helps to maintain scalability in the number of source services.

Accordingly, the publication operation returns two time stamps `TS4` and `TS5`, which we call *minimum idle time* and *maximum idle time*, respectively. The semantics of the minimum idle time are as follows: *"Publication was successful, but in the future you may be dropped and denied service if you do not wait at least until time* `TS4` *before the next refresh"*. The semantics of the maximum idle time are as follows: *"Publication was successful, but in the future you may be dropped and denied service if you do not refresh before time* `TS5` *"*. We have `minimum idle time < maximum idle time`. A simple hyper registry always returns zero and infinity as minimum idle and maximum idle time, respectively. Content providers ignoring throttling warnings can be dropped and denied service without further notice, for example at the local, firewall or Internet Service Provider (ISP) level. Analogously, query operations return a minimum idle time (`TS4`) as part of the result set. The semantics are as follows: *"The query was successful, and here is the result set. However, in the future you may be dropped and denied service if you do not wait at least until time* `TS4` *before the next query"*.

7. Related Work.

RDBMS. Relational database systems [24] provide SQL as a powerful query language. They assume tight and consistent central control and hence are infeasable in Grid environments, which are characterized by heterogeneity, scale, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. They do not support an XML data model and the XQuery language. The relational data model does not allow for semi-structured data. A query must have out-of-band knowledge of the relevant table names and schemas, which themselves may not be heterogeneous but must be static, globally standardized and synchronized. This seriously limits the applicability of the relational model in the context of autonomy, decentralization, unreliability and frequent change. SQL lacks hierarchical navigation as a key feature and other capabilities such as dynamic data integration, leading to extremely complex queries over a large number of auxiliary tables [25]. Further, RDBMS do not provide dynamic content generation, soft state based publication and content caching. Our work does not compete with an RDBMS, though. A registry may well internally use an RDBMS for data management. A registry can accept queries over an XML view and internally translate the query into SQL [26, 25]. The relational data model and SQL are, for example, used in the Relational Grid Monitoring Architecture (RGMA) system [27] and the Unified Relational GIS Project [28].

Web Proxy Caches. A weak cache coherency policy popular with web proxy caches is *adaptive TTL* [20]. Here the problem is handled by adjusting the time-to-live of a content based on observations of its lifetime. Adaptive TTL takes advantage of the fact that content lifetime distribution tends to be bimodal; if a given content has not been modified for a long time, it tends to stay unchanged. Thus, the time-to-live attribute of a given content is assigned to be a percentage of the content's current "age", which is the current time minus the last modified time of the document.

The "web server accelerator" [29] resides in front of one or more web servers to speed up user accesses. It provides an API, which allows application programs to explicitly add, delete, and update cached data. The API allows the accelerator to cache dynamic as well as static data. Invalidating and updating cached data is facilitated by the Data Update Propagation (DUP) algorithm, which maintains data dependence information between cached data and underlying data in a graph [30].

ANSA and CORBA. The ANSA project was an early collaborative industry effort to advance distributed computing. It defined trading services [31] for advertizement and discovery of relevant services, based on service type and simple constraints on attribute/value pairs. The CORBA Trading service [32] is an evolution of these efforts.

UDDI. UDDI (Universal Description, Discovery and Integration) [33] is an emerging industry standard that defines a business oriented access mechanism to a registry holding XML based WSDL [8] service descriptions. It is not designed to be a registry holding arbitrary content. UDDI is not based on soft state, which implies that there is no way to dynamically manage and remove service descriptions from a large number of autonomous third parties in a reliable, predictable and simple way. It does not address the fact that services often fail or misbehave or are reconfigured, leaving a registry in an inconsistent state. Content freshness is not addressed. As such, UDDI only appears to be useful for businesses and their customers running static high availability services. Last, and perhaps most importantly, query support is rudimentary. Only key lookups with primitive qualifiers are supported, which is insufficient for realistic service discovery use cases (see Figure 3.4 for examples).

Jini, SLP, SDS, INS. The centralized Jini Lookup Service [34] is located by Java clients via a UDP multicast. The network protocol is not language independent because it relies on the Java-specific object serialization mechanism. Publication is based on soft state. Clients and services must renew their leases periodically. Content freshness is not addressed. The query "language" allows for simple string matching on attributes, and is even less powerful than LDAP.

The Service Location Protocol (SLP) [35] uses multicast, softstate and simple filter expressions to advertize and query the location, type and attributes of services. The query "language" is more simple than Jini's. An extension is the Mesh Enhanced Service Location Protocol (mSLP) [36], increasing scalability through multiple cooperating directory agents. Both assume a single administrative domain and hence do not scale to the Internet and Grids.

The Service Discovery Service (SDS) [37] is also based on multi cast and soft state. It supports a simple XML based exact match query type. SDS is interesting in that it mandates secure channels with authentication and traffic encryption, and privacy and authenticity of service descriptions. SDS servers can be organized in a distributed hierarchy. For efficiency, each SDS node in a hierarchy can hold an index of the content of its

sub-tree. The index is a compact aggregation and custom tailored to the narrow type of query SDS can answer. Another effort is the Intentional Naming System [38]. Like SDS, it integrates name resolution and routing.

LDAP. The Lightweight Directory Access Protocol (LDAP) [39] defines an access mechanism in which clients send requests to and receive responses from LDAP servers. The data model is not based on soft state. Content freshness is not addressed. LDAP does not follow an XML data model. An LDAP query is an expression that logically compares (`=, <=, >=`) the string value of an attribute (`email`) with a string constant, optionally with a substring match joker (`picture*.jpg`) and approximate string equality test (∼). Expressions can be combined with Boolean `AND, OR` and `NOT` operators. The expressive power of the LDAP query language is insufficient for service discovery use cases (see Figure 3.4 for examples) and most other non-trivial questions.

MDS. The Metacomputing Directory Service (MDS) [40, 41] is based on LDAP. As a result, its query language is insufficient for service discovery, and it does not follow an XML data model. MDS is based on soft state but it does not allow clients (and to some extent even content providers) to drive registry freshness policies.

The MDS consists of an unmodified OpenLDAP [42] server with value-adding backends, configured with a strong security library. The Grid Resource Information Service (GRIS) is an OpenLDAP backend that accepts LDAP queries from clients over its own LDAP namespace sub-tree. It is a backend into which a list of content providers can be plugged on a per attribute basis. An example content provider is a shell script or program that returns the operating system version. A provider owns a namespace sub tree of the GRIS and returns a set of LDAP entries within that namespace. Depending on the namespace specified in an LDAP query, the GRIS executes (and creates the processes for) one or more affected providers and caches the results for use in future queries. It then applies the query against the cache. A GRIS can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). An example GRIS configuration invokes the `/usr/local/bin/grid-info-cpu-linux` executable and caches CPU load results for 15 seconds. Content provider invocation follows a CGI like life cycle model. The stateless nature, heavy weight process forking and context switches of such a model render it unsuitable for use in dynamic environments with high frequency refreshes and requests [43].

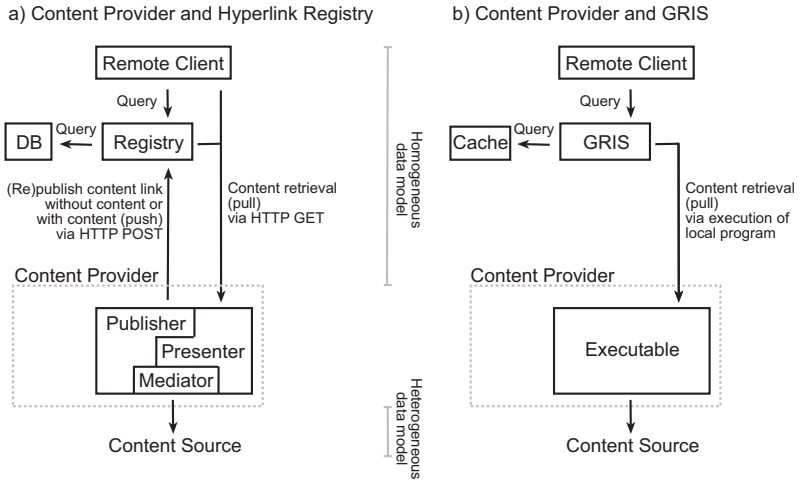


FIG. 7.1. *Hyper Registry and Grid Resource Information Service.*

Figure 7.1 contrasts the different architectures of a GRIS and a hyper registry. A hyper registry maintains content links and cached content in its database, whereas a GRIS maintains cached content only. The control paths from client to content provider and from content provider to the hyper registry are missing in the GRIS architecture, disabling cache freshness steering. A GRIS content provider is always local to the GRIS and cannot publish to a remote GRIS. In contrast, a content provider is cleanly decoupled from a hyper registry and only requires the ubiquitous HTTP protocol for simple communication with a local or remote registry. A GRIS requires implementing the complex LDAP protocol, including its query language, at every content provider location. In contrast, handling the powerful but complex XQuery language is only required within a

hyper registry, not at the content provider. Table 7.1 summarizes some commonalities and differences related to publication and content freshness.

TABLE 7.1
Comparison of Hyper Registry and Metacomputing Directory Service.

Question	**MDS**	**Hyper Registry**
Can a provider publish to a remote registry?	No. A provider is local to a GRIS.	Yes. A provider can publish to any hyper registry, no matter whether the hyper registry is deployed locally, remotely or in-process.
Can a provider actively steer registry freshness?	No. A GRIS is actively pulling content. It can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). A content provider is passive. It cannot actively publish and refresh content and hence cannot steer the freshness of its content cached in the GRIS.	Yes. Content provider and hyper registry are active and passive at the same time. At any time, a hyper registry can actively pull content, and a content provider can actively push with or without content. Both components can steer the freshness of content cached in the hyper registry.
Can a client retrieve current content?	No. A client cannot retrieve the current content from a content provider. It has to go through a GRIS or GIIS, which normally return stale content from their cache.	Yes. A client can directly connect to a content provider and retrieve the current content, thereby avoiding stale content from a hyper registry.
Can a client query steer result freshness?	No. A client query cannot steer the freshness of the results it generates.	Yes. A client query can steer the freshness of the results it generates via the refresh-on-client-demand strategy.

8. Conclusions. We address the problems of maintaining and querying dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. The hyper registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. A content provider is cleanly decoupled from the hyper registry and only requires the ubiquitous HTTP protocol for communication with a local or remote hyper registry.

As future work, it would be interesting to study and specify in more detail specific cache freshness interaction policies between content provider, hyper registry and client (query). Our specification allows expressing a wide range of policies, some of which we outline, but we do not evaluate in detail the merits and drawbacks of any given policy.

REFERENCES

[1] LARGE HADRON COLLIDER COMMITTEE, *Report of the LHC Computing Review,* Technical report, CERN/LHCC/2001-004, April 2001, `http://cern.ch/lhc-computing-review-public/Public/Report_final.PDF`

[2] BEN SEGAL, *Grid Computing: The European Data Grid Project,* In IEEE Nuclear Science Symposium and Medical Imaging Conference, Lyon, France, October 2000.

[3] WOLFGANG HOSCHEK, JAVIER JAEN-MARTINEZ, ASAD SAMAR, HEINZ STOCKINGER AND KURT STOCKINGER, *Data Management in an International Data Grid Project.* In 1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000), Bangalore, India, December 2000.

[4] Ian Foster, Carl Kesselman and Steve Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations,* Int'l. Journal of Supercomputer Applications, 15(3), 2001.
[5] Wolfgang Hoschek, *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery,* PhD Thesis, Technical University of Vienna, March 2002.
[6] Ian Foster, Carl Kesselman, Jeffrey Nick and Steve Tuecke, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, January 2002, `http://www.globus.org.`
[7] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev, *Professional XML Web Services.* Wrox Press, 2001.
[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1.* W3C Note 15, 2001, `http://www.w3.org/TR/wsdl`
[9] World Wide Web Consortium, *XQuery 1.0: An XML Query Language,* W3C Working Draft, December 2001.
[10] World Wide Web Consortium, *XML Query Use Cases* W3C Working Draft, December 2001.
[11] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany, *A Grid Monitoring Architecture,* Technical report, Global Grid Forum Informational Document, January 2002, `http://www.gridforum.org`
[12] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax,* IETF RFC 2396.
[13] World Wide Web Consortium, *XML Schema Part 0: Primer,* W3C Recommendation, May 2001.
[14] Apache Software Foundation, *The Jakarta Tomcat Project,* `http://jakarta.apache.org/tomcat/`
[15] World Wide Web Consortium, *XML-Signature Syntax and Processing,* W3C Recommendation, February 2002.
[16] P. Brittenham, *An Overview of the Web Services Inspection Language,* 2001, `www.ibm.com/developerworks/webservices/library/ws-wsilover`
[17] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,* IETF RFC 2045, November 1996.
[18] International Organization for Standardization (ISO), *Information Technology-Database Language SQL,* Standard No. ISO/IEC 9075:1999, 1999.
[19] Software AG, *The Quip XQuery processor,* `http://www.softwareag.com/developer/quip/`
[20] J. Wang, *A survey of web caching schemes for the Internet,* ACM Computer Communication Reviews, 29(5), October 1999.
[21] S. Gullapalli, K. Czajkowski, C. Kesselman, and S. Fitzgerald, *The grid notification framework,* Technical report, Grid Forum Working Draft GWD-GIS-019, June 2001, `http://www.gridforum.org`
[22] C. Weider, A. Herron, A. Anantha, and T. Howes, *LDAP Control Extension for Simple Paged Results Manipulation,* IETF RFC 2696.
[23] M. P. Maher and C. Perkins, *Session Announcement Protocol: Version 2,* IETF Internet Draft draft-ietf-mmusic-sap-v2-00.txt, November 1998.
[24] M. Tamer Özsu and Patrick Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, 1999.
[25] Daniela Florescu, Ioana Manolescu, and Donald Kossmann, *Answering XML Queries over Heterogeneous Data Sources,* In Int'l. Conf. on Very Large Data Bases (VLDB), Roma, Italy, September 2001.
[26] Mary Fernandez, Morishima Atsuyuki, Dan Suciu, and Tan Wang-Chiew, *Publishing Relational Data in XML: the SilkRoute Approach,* IEEE Data Engineering Bulletin, 24(2), 2001.
[27] Steve Fisher et al, *Information and Monitoring (WP3) Architecture Report,* Technical report, DataGrid-03-D3.2, January 2001.
[28] W. P. Dinda and B. Plale, *A Unified Relational Approach to Grid Information Services.* Technical report, Grid Forum Informational Draft GWD-GIS-012-1, February 2001, `http://www.gridforum.org`
[29] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias, *Design and performance of Web server accelerator.* In Proceedings of Infocom'99, 1999.
[30] J. Challenger, A. Iyengar, and P. Dantzig, *A scalable system for consistently caching dynamic Web data,* In Proceedings of Infocom'99, 1999.
[31] Ashley Beitz, Mirion Bearman, and Andreas Vogel, *Service Location in an Open Distributed Environment,* In Proc. of the Int'l. Workshop on Services in Distributed and Networked Environements, Whistler, Canada, June 1995.
[32] Object Management Group, *Trading Object Service,* OMG RPF5 Submission, May 1996.
[33] UDDI Consortium, *UDDI: Universal Description, Discovery and Integration,* `http://www.uddi.org`
[34] J. Waldo, *The Jini architecture for network-centric computing,* Communications of the ACM, 42(7), July 1999.
[35] Erik Guttman, *Service Location Protocol: Automatic Discovery of IP Network Services,* IEEE Internet Computing Journal, 3(4), 1999.
[36] Weibin Zhao, Henning Schulzrinne, and Erik Guttman, *mSLP—Mesh Enhanced Service Location Protocol.* In Proc. of the IEEE Int'l. Conf. on Computer Communications and Networks (ICCCN'00), Las Vegas, USA, October 2000.
[37] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz, *An Architecture for a Secure Service Discovery Service,* In Fifth Annual Int'l. Conf. on Mobile Computing and Networks (MobiCOM '99), Seattle, WA, August 1999.
[38] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley, *The design and implementation of an intentional naming system.* In Proc. of the Symposium on Operating Systems Principles, Kiawah Island, USA, December 1999.
[39] W. Yeong, T. Howes, and S. Kille, *Lightweight Directory Access Protocol,* IETF RFC 1777, March 1995.
[40] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman, *Grid Information Services for Distributed Resource Sharing,* In Tenth IEEE Int'l. Symposium on High-Performance Distributed Computing (HPDC-10), San Francisco, California, August 2001.
[41] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke, *A Directory Service for Configuring High-Performance Distributed Computations,* In 6th Int'l. Symposium on High

Performance Distributed Computing (HPDC '97), 1997.
[42] The OpenLDAP project, *The OpenLDAP project,* `http://www.openldap.org`
[43] A. Wu, H. Wang, and D. Wilkins, *Performance Comparison of Web-To-Database Applications,* In Proceedings of the Southern Conference on Computing, The University of Southern Mississippi, October 2000.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: August 12, 2002
Accepted: December 13, 2002

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 17–32. http://www.scpe.org
ISSN 1895-1767

AD HOC METACOMPUTING WITH COMPEER

KEITH POWER* AND JOHN P. MORRISON*

Abstract.
Metacomputing allows the exploitation of geographically seperate, heterogenous networks and resources. Most metacomputers are feature rich and carry a long, complicated installation, requiring knowledge of accounting procedures, access control lists and user management, all of which differ from system to system. Metacomputers can have high administrative overhead, and a steep learning curve which restricts their utility to organisations which can afford these costs.

This paper describes the Compeer system, which attempts to make metacomputing more accessible by employing an implicitly parallel computing model, support for programming this model with a Java-like language and the construction of a dynamic *ad hoc* metacomputer that can be temporarily instantiated for the purpose of executing applications.

Key words. Peer-to-peer, decentralised, metacomputer, ad hoc metacomputing, graph, java, translation, condensed graphs

1. Introduction. The variety of metacomputers has grown considerably in recent years. There are batch processing systems, like CODINE [21] and PBS [7]. There are metacomputers designed with specific types of application in mind, like DiscWorld [20] and AppLes [2]. There are toolkits to facilitate building tailored distributed applications, like Globus [5], and there are systems which present a network as a single virtual machine (VM), like Legion [13]. There is even a system, Everyware [22], which links these metacomputers together.

These systems have tremendous flexiblity of operation, allowing different sites to link and share resources, to store detailed accounting information to ensure resources are shared equally, or to charge an organisation for its usage. These features are a consequence of the large user base and the permanence of the associated systems.

Exposure to metacomputing systems comes also in the form of popular metacomputing applications such as SETI@Home[19] and distributed.net [4]. Configuration and setup costs are kept to a minimum using a simple download and installation procedure. From the participant's perspective, CPU cycles are contributed to solving a single, long running application. Since compute tasks cannot be submitted by a participant to the metacomputer, the benefit of these systems is unidirectional, relying on the altruism of the volunteers.

Compeer does not have any preexisting, or necessarily permanent, architectural component or communications topology. Rather, it is dynamically constructed in an ad hoc manner by the coming together of multiple "peer" machines. Peers join the collective through either a process of discovery or introduction.

The advantage of the Compeer system comes from the fact that it is easily and inexpensively deployed similarly to the popular metacomputing applications, yet it is a metacomputing platform giving each peer the facility to contribute to and benefit from the association. The remaining sections of this paper will discuss the Compeer program methodology, an overview of the Compeer architecture and possible deployments and finally present two sample applications.

Compeer executes applications in the form of Condensed Graphs ($\mathcal{CG}$s). The $\mathcal{CG}$ model of computing is a very expressive, implicitly parallel model which allows different sequencing constraints to be specified in applications. It provides automatic synchronisation, and is hierarchical, facilitating the easy distribution of work across a network. The sequencing facilities of $\mathcal{CG}$s are not relevant here, so for the purposes of this paper $\mathcal{CG}$s can be considered as similar to Dataflow [9] graphs.

Applications can be translated from Java[6] to an equivalent $\mathcal{CG}$ form to provide automatic parallelism. Java was chosen because it has a large user base and a well defined, publicly available grammar. An Object Oriented language was chosen to facilitate the easy integration of objects into $\mathcal{CG}$ applications

2. Overview of Compeer.

2.1. Architecture.

Compeer is a Java based peer-to-peer metacomputing platform. Each peer in the metacomputer is a Compeer object. These objects, linked together, form the metacomputer. It is possible to have more than one Compeer object running per machine, but for the purpose of this paper, it is assumed that each machine runs a single Compeer object, and so the term peer is applied interchangeably to both a Compeer object and the machine on which it resides. Being peer-to-peer, each peer is equal in ability and communicates symmetrically with other peers. Since the system is serverless, there is no single point of administration,

*Computer Science Dept., University College Cork, Ireland (k.power@cs.ucc.ie, j.morrison@cs.ucc.ie)

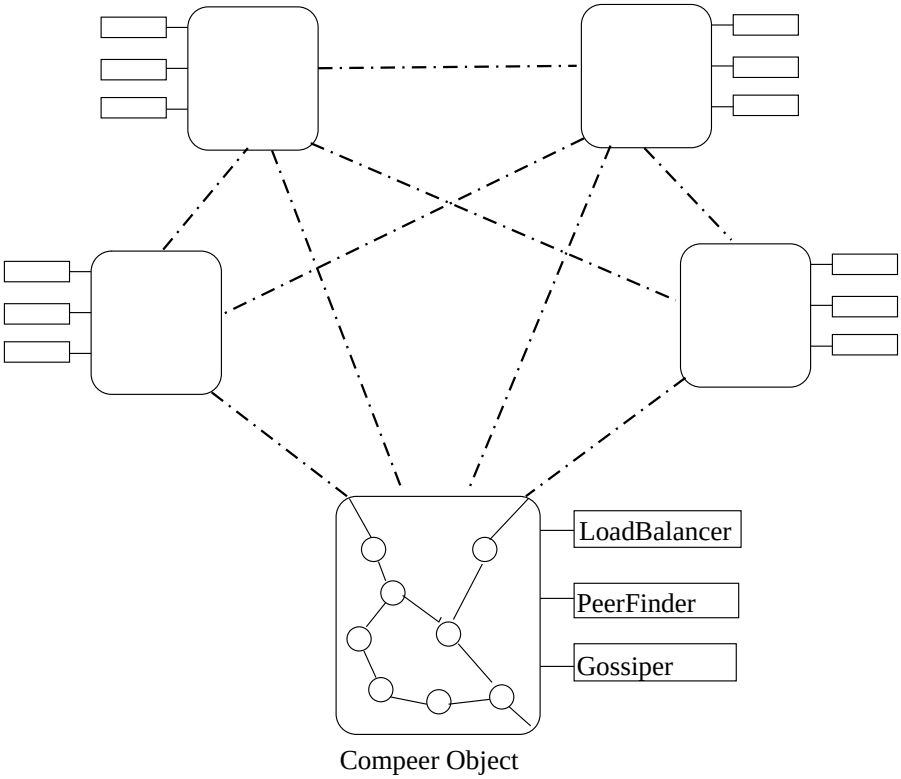


FIG. 2.1. The Compeer Architecture. A system of 5 peers is illustrated with $\mathcal{CG}$s running on each peer and communicating with each other. The Compeer object on each peer is responsible for gathering system information, for creating objects corresponding to incoming $\mathcal{CG}$s and for initiating the creation of $\mathcal{CG}$s on other peers.

which enables the construction of cooperative metacomputers by willing participants facilitating the sharing of resources.

Each Compeer object is supplied with several helper objects at startup: a PeerFinder object, a LoadBalancer object and a Gossiper object. The PeerFinder object is responsible for discovering other peers and keeping references to them. The LoadBalancer object is responsible for deciding where in the network new nodes should be created, based on a list of peers and their relevant statistics e.g. CPU load. The Gossiper object is responsible for inter-peer communications, for example, spreading results around the network when a computation is complete, which also requires a list of peers. Three java interfaces are used, rather than classes, facilitating the easy replacement of any of the components.

This modular approach yields a versatile metacomputing system that can be used in a variety of deployments, from intra- to inter- net. For example, in an intranet deployment where the number of machines is small, a PeerFinder which discovers other peers via multicast is viable and can be used. Over the Internet, a PeerFinder which reads the IPs of other peers from a webpage can be implemented. In small deployments, each peer can hold a reference to all other peers in the network, and so a very simple Gossiper can be used to propogate messages. With a large network, each peer may contain references to only a portion of the other peers in order to allow scaling. This would require a more complex Gossiper.

Compeer metacomputer can also be constructed in an ad hoc fashion, with peers connecting and leaving the network at will. This method makes several useful deployments possible, for example a situation where several researchers run their own peers and direct them to the other peers, allowing them to share their resources.

2.2. Program Execution. Compeer executes programs in the form of $\mathcal{CG}$s, described in a graph description language. $\mathcal{CG}$s are hierarchical, so a single application consists of descriptions of multiple graphs. As a graph executes, each node in the graph is represented by a CGNode Java object in Compeer. Specific nodes such as the Filter node are subclasses of an abstract class, CGNode. This nodes implements the $\mathcal{CG}$ firing rule, which means that a node which has its required inputs in the correct form and has a destination can fire. In

this implementation, resources permitting, any node that can fire, fires immediately. $\mathcal{CG}$ nodes in the graph are represented in Compeer by a $\mathcal{CG}$ object, which contains a variable detailing the KDLang instructions it will use when firing. When a $\mathcal{CG}$ node fires, it's instructions are interpreted, leading to the creation and linking together of nodes representing that $\mathcal{CG}$. This permits a simple approach to program execution: create a single CG object, supply it with the graph instructions and give it an X node as a destination. The CG node will fire to create the objects corresponding to the nodes in a graph and link them together as represented by the graph. Any fireable nodes will fire, placing their results on the input ports of other nodes, causing them to fire and so on, until a result is produced and fired through the Exit node ending the computation

Nodes can be created on the local peer or on any other known peer. The decision of where to create them is the responsibility of the LoadBalancer object. A peer can run many applications simultaneously, and the nodes in each application are separate and comprise a *process*. Before a peer can send nodes for a process to other peers it must 'introduce' those peers to the process by sending them the graph description of the application being executed.

2.3. Primitive operations in Compeer. Earlier implementations of the $\mathcal{CG}$ platform used functions as operators, or primitive nodes, to accomplish data transformation e.g. the plus node in (Fig. 4.1) corresponds to a plus function at execution time.

In Compeer data transformation is accomplished via method invocations on objects. Correspondingly, Compeer requires nodes which can represent the creation of these objects, and invocations upon them. Java, DCOM and CORBA objects are currently supported via Compeer plugins, though plugins to support other object technologies can be developed. Basic type conversion between different object technologies is handled by Compeer.

To support objects as node primitives, as opposed to functions like `+` in the factorial example, Compeer introduces two nodes, `create` and `invoke`.

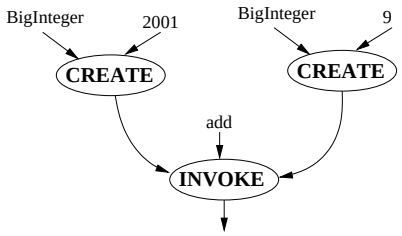


FIG. 2.2. Example of `create` and `invoke` nodes

The `create` node is used to dynamically create any Java object. It takes the name of the class to create as a parameter, and optionally, parameters for a constructor. There are also `createDCOM` and `createCORBA` nodes, although it is also possible now to create CORBA objects using the libraries supplied with the Java Development Kit 1.4.

The `invoke` node is used to dynamically invoke methods on an object. It takes an object reference, a method name and optionally parameters. In cases of method overloading the method to execute is determined using the parameter types. Similarly, there are `invokeDCOM` and `invokeCORBA` nodes.

In (Fig. 2.2), both `create` nodes will create instances of the `BigInteger` class and initialise them with 2001 and 9. The `add` method on one of these instances will be invoked using the other as a parameter.

The $\mathcal{CG}$s that Compeer executes consist of nodes representing the creation and execution of Java, CORBA and DCOM objects. In fact, Compeer can be viewed simply as a distributed scheduler of these objects.

2.4. Reflection and Polymorphic nodes. A Compeer object can itself be manipulated by a graph it is executing. This possibility arises because Compeer is implemented as a Java object. A graph which obtains a reference to the Compeer on which it is running can invoke methods on it to obtain information or manipulate other graphs. This is useful for cases where an application needs to know how many peers are connected to the peer it's running on, or how busy a peer is.

Nodes in Compeer are also objects, therefore they can be manipulated in graphs too. This can be used to set a $\mathcal{CG}$ nodes instructions at run time. A node whose instructions are altered during the course of its existence is termed *polymorphic*.

Programmatic dynamic load balancing can be achieved by combining these two techniques. For example, if an application consisting of 100 resource intensive $\mathcal{CG}$s were executed on a single peer and each of the individual $\mathcal{CG}$s were handed out to peers, then the first peer must distribute 100 nodes itself. This is analogous to the master/slave concept of work distribution. If, at run time, the graph tailored it's behaviour to the number of peers available, better results could be achieved. For example, after interrogating the Compeer object and discovering there are 5 peers connected whose loads are low enough to be useful, instructions could be produced for a node that when fired would distribute a $\mathcal{CG}$ containing 20 of the tasks. This process could be repeated on those peers to facilitate the rapid distribution of work and take advantage of the topology of the network.

This technique allows the construction of $\mathcal{CG}$ applications which can tailor their behaviour to the underlying architecture at run time.

It is also possible for Compeer to alter the description of the graph it is executing to optimise performance. For example, an idle Compeer could alter certain connections in the graph to make more instructions eager, and thus take more advantage of the CPU. Another example would be where Compeer measures the execution of a method invocation and alters the graph to reflect the measurement. In this way feedback from the executing graph can alter it's future execution. Only the order and timing of execution is changed, the program remains the same.

3. Examples of Deployment. Compeer can be deployed in a variety of situations, due to it's minimal configuration, decentralized architecture and object based interface. Three such deployments are presented now.

Compeer is completely decentralized and therefore has no single point of administration. While a single point of administration is generally considered advantageous in networked systems, the lack of one here allows the metacomputer to be deployed in an ad hoc fashion, that is, the metacomputer can be formed by peers, located anywhere, joining and leaving at any time. This allows a cooperative metacomputer to be built among willing users, where each user runs their own peer on their own machine. To connect into the network they need know only the IP of one other peer, from which their Compeer can obtain references to other peers. Since Compeer is Java based, each user can customise their security policies to enable as much sharing of resources as they wish, so there is no need for a global administrator of the system. This type of arrangement enables researchers to construct applications capable of easily sharing data and resources with their peers.

To deploy Compeer it is sufficient to supply each peer with a list of IP/name pairs it can use to find other peers. These can be supplied as a file when each Compeer is executed, or placed on a web page where they can be downloaded. This simple installation enables another type of ad hoc deployment, where a single user can start a Compeer on multiple machines to form a usable metacomputer. This is useful in cases where a user has an application which takes a lot of CPU time which they want to finish sooner. Rather than setting up and configuring a metacomputer, which may take longer than the application, they can deploy Compeer quickly and execute their application immediately. If necessary, Compeer can automatically uninstall when the application ends. Compeer can be started and stopped quickly and easily, reducing the cost of investment associated with setting up a metacomputer.

Each peer is an object, which makes Compeer versatile in terms of interfacing. An example deployment which exploits this would place peers on a network in any arrangement, including the two previously described, but would be accessed via a webserver. An internet user could visit a webpage which is generated dynamically, for example, a Java Server Page or Servlet. This page could interface with any Compeer in the network, submit some work, retrieve the result, parse it and send it to the users web browser to display. This enables users to easily construct web based interfaces to their network, or to distribute the load of their webserver across their network.

The system is written in Java, so it can operate on Windows, *nix and any other operating system for which a JVM is available. It can be configured to run in the background on *nix systems, or as a service on Windows, to permanently install it.

3.1. Note on security. The system leverages operating system security and a simple password system to prevent unauthorised users submitting work to the metacomputer, or receiving work or results from it.

Where one user has configured the system, Compeer objects running on each machine execute in that users process space. Where several people set up the system, there is a Compeer object running on each of their machines, in their process space. It is assumed that these people trust each other. In cases where trust is not

absolute, individual users can tailor their *java.policy* file to restrict other users access to their machine e.g., to prevent file system access and/or to deny loading of DLLs. The operating systems separation of process space prevents unauthorised users from tampering with the running Compeer objects.

It is possible for malicious users of the system to produce fake results. Currently, no effort is made to detect this or protect against it. Several methods of protection in a master/slave arrangment have been put forward [17]. The problem is compounded in a peer-to-peer metacomputer. One approach to sabotage tolerance is to distribute multiple copies of a piece of work to different slaves. If the slaves return different results, then one of them must be falsifying it's data, and it's identity can be ascertained by further investigation. Applying this approach in Compeer situation is infeasible: a peer passing a $\mathcal{CG}$ to one peer could instead pass duplicates to several peers. Any $\mathcal{CG}$s in that application will also be duplicated by each peer, and so on, leading to an exponential increase in the amount of work to be done at each stage, rendering the distributed execution of any non-trivial application counterproductive.

Digital signatures have been used to verify the identity of participants [18]. A method of producing an audit trail for computations using digital signing is being investigated. It is envisaged that this will allow the identities of peers who produce incorrect results, enabling a punishment policy, such as blacklisting, to be introduced.

3.2. Usage. Compeer is completely decentralised, and so jobs can be submitted to any peer in the network for execution. The metacomputer has a versatile interface enabled by the fact that each peer is an object, and each peer is equal. Once a reference to any peer is obtained, currently via RMI, the peer can be manipulated, examined or sent work. It is also possible to obtain references to its known peers, and so on, until a complete picture of the network is built up. This ease of manipulation leads to a wide variety of usage scenarios. Work can be submitted to a peer via a graphical or command line shell. The metacomputer can be invoked from a JSP to provide dynamic content for webpages, or invoked using XML/SOAP to enable web services. GUIs can be built whose buttons result in the execution of complex distributed applications across a network via a single, simple method call.

For non-programmatic access, a user interfaces with the Compeer metacomputer via a graphical shell. A shell can connect to any Compeer object, though in practice it is more efficient to connect to the Compeer object running locally. A text file representing a $\mathcal{CG}$ application is loaded using the shell and is submitted to the system. If debugging is disabled, the application executes quietly until a result is returned.

If debugging is enabled, a list of peers participating in this process is displayed in the shell. For each peer, a list of any of the current process's nodes present on that machine is listed. These nodes can be selected to highlight their inputs and destinations. The computation can proceed forward in a step-by-step fashion. A global step informs all Compeer objects to execute the next instruction available in this process. An individual step directs a single Compeer object to execute its next available instruction for this process. Alternatively, individual nodes can be fired and their properties inspected. Thus, per-process, per-peer and per-node debugging is possible. Debugging a process has no effect on any other executing Compeer processes.

Debugging is useful to users with knowledge of $\mathcal{CG}$s. Knowledge of $\mathcal{CG}$s and how they execute in Compeer is not a prerequisite for using this system, since Java programs can be converted into suitable graphs, though it is recommended for users interested in optimising their applications performance.

3.3. Note on Application Development. $\mathcal{CG}$ applications are written in KDLang, a simple language used for constructing graphs. They can be converted to XML for use with other graph based applications, in particular [14]. Development of these applications using KDlang can be time-consuming and error-prone. To ease this process, programs can be developed graphically using a drag-and-drop development environment, or via a visual KDLang editor (Fig. 3.1), which displays the graph as it's being described. Large applications can be produced by converting Java to KDLang, and then optionally tailoring the generated code in the visual editor.

4. KDLang grammar. The KDLANG language consists of instructions for building condensed graphs. There are instructions to create nodes, to connect them in various ways (either a stem or graft to facilitate eager or lazy execution), and instructions to set the value of simple nodes. An example of the instructions needed to create the graph in (Fig. 4.1) follows:

```
CREATE PLUS addNode
CREATE SIMPLE a
CREATE SIMPLE b
```

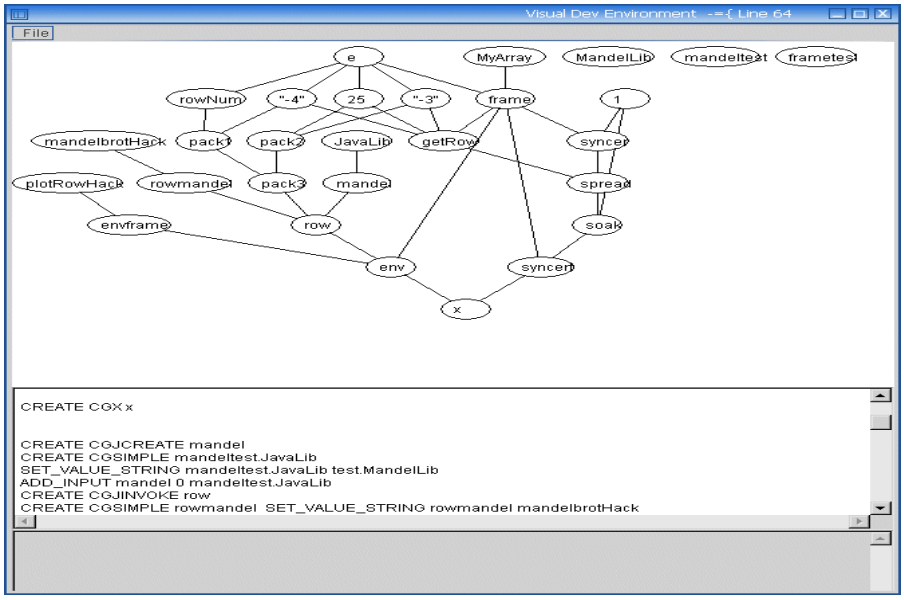


FIG. 3.1. Visual Development Environment. As the KDLang graph description is entered into the center panel, the graph is constructed and displayed in the top panel. Any errors in the KDLang are reported in the bottom panel.

```
CREATE X result
SET_VALUE_INTEGER a 5
SET_VALUE_INTEGER b 7
ADD_INPUT addNode 0 a
ADD_INPUT addNode 1 b
ADD_DESTINATION addNode result 0
```

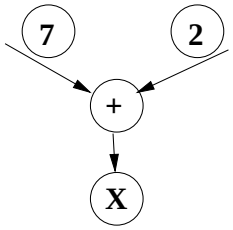


FIG. 4.1. *Simple CG to add two integers. Written without Create and Invoke nodes.*

4.1. Definition of KDLang. The KDLang grammar is defined using the ANTLR[15] syntax rules.

```
class P extends Parser;

programDef
   :  (cgDef)+ EOF
   ;
cgDef
   :  WORD NUMBER (statement)* CGEND
   ;
statement
   :  (createOp | addInputOp | addDestOp | setValOp)
   ;
createOp
   :  createSym WORD WORD
```

```
   ;
addInputOp
   :  addInputSym WORD NUMBER WORD
   ;
addDestOp
   :  addDestSym WORD WORD NUMBER
   ;
setValOp
   :  ( setValIntOp | setValStringOp | setValDoubleOp
                    | setValCharOp   | setValBoolOp   )
   ;
setValIntOp
   :  setValIntSym WORD NUMBER
   ;
setValCharOp
   :  setValCharSym WORD WORD
   ;
setValBoolOp
   :  setValBoolSym WORD WORD
   ;
setValDoubleOp
   :  setValDoubleSym WORD NUMBER
   ;
setValStringOp
   :  setValStrSym WORD WORD
   ;

createSym:              "CREATE"
   ;
addInputSym:    "ADD_INPUT"
   ;
addDestSym:             "ADD_DESTINATION"
   ;
setValIntSym:   "SET_VALUE_INTEGER"
   ;
setValCharSym:  "SET_VALUE_CHARACTER"
   ;
setValBoolSym:  "SET_VALUE_BOOLEAN"
   ;
setValStrSym:   "SET_VALUE_STRING"
   ;
setValDoubleSym:"SET_VALUE_DOUBLE"
   ;

////////////////// Lexer

class L extends Lexer; // one-or-more letters followed by a newline

CGEND:  '#'
    ;
protected
LETTER: ('a'..'z'|'A'..'Z'|'_'|'.' )
    ;
protected
```

```
DIGIT:  ('0'..'9')
    ;
WORD:   LETTER (LETTER | DIGIT )*
    ;
NUMBER: ('-'|'+')? (DIGIT)+ ('.' (DIGIT)+ )?
    ;
// Whitespace -- ignored
WS      :       (       ' '
                |       '\t'
                |       '\f'
                        // handle newlines
                |       (       options {generateAmbigWarnings=false;}
                        :       "\r\n"  // DOS
                        |       '\r'    // Macintosh
                        |       '\n'    // Unix
                        )
                )+
                { _ttype = Token.SKIP; }
        ;
```

Although the graph in (Fig. 4.1) is a simple graph, it still requires 9 KDLang instructions to build it. To build a corresponding graph using objects and method invocations, like that in (Fig. 2.2) takes more than double this amount. The KDLang language, while suited to easy interpretation by Compeer, is cumbersome when used to develop non-trivial applications. For this purpose, Java code is converted into KDLang, which can then be optimised or executed on Compeer as is. To illustrate the suitability of using Java in place of KDLang for describing object creation and invocation, a code snippet corresponding to (Fig. 2.2) is presented:

```
(new BigInteger("2001")).add(new BigInteger("9"))
```

A detailed description of the process involved in translating Java to an equivalent form of $\mathcal{CG}$s follows. For clarity, images of the corresponding graphs are shown, rather than KDLang examples.

5. Translation Process.

5.1. Motivation. Most single-threaded programs contain a surprising amount of implicit parallelism [10],[11]. This parallelism can be automatically uncovered by converting Java to equivalent $\mathcal{CG}$ graphs, and executing them on the Compeer platform. Using this approach, distributed applications can be produced from single-threaded code without the need for RMI[8] or message passing libraries, such as MPJ[3]. The consequent application deployment is also simpler since the programmer is not responsible for distributing the application across the network or for managing proxy and stub objects.

In contrast, tools such as JavaParty[16] and Do![12] exist which distribute multithreaded Java applications exploiting programmer designed parallelism across a network, in what is known as a "nearly transparent" manner. That is, the tools require that the programmer make only small changes to the application code before it is run. The burden of ensuring synchronisation and concurrency, however, still remains with the programmer.

This work is motivated by the fact that many types of programs lend themselves to the type of conversion described here. These include, parameter sweep applications, key cracks and image processing algorithms.

5.2. ANTLR. ANTLR[15] was used to implement this process because it uses a uniform language for specifying the lexical and parsing rules which make up a grammar. A Java grammar is freely available for ANTLR, which facilitates easy generation of a parser that can recognise all Java constructs.

Action code, in the form of Java snippets, is embedded in a grammar rule wherever an action is desired. For example, to print out the names of all variables in a Java program, a line of code to print out the current token would be added to the rule which recognises variable declarations. To effect translation from Java to KDLang, action code is inserted into relevant grammar rules which outputs the equivalent KDLang to a file.

5.3. Caveats. Note that the Java program to be converted into a $\mathcal{CG}$ should compile and run correctly on a single machine before conversion is attempted, since this translation process performs no type or syntax checking.

Only the class containing the main method is converted. Each of the methods in this class is converted into a $\mathcal{CG}$. The main method is converted into a $\mathcal{CG}$ which uses these other $\mathcal{CG}$s. This gives the programmer control over the grain size of instructions to be distributed. If all involved classes were converted to $\mathcal{CG}$s, the resulting grain size would be so fine as to slow down the computation.

Consider, for example, a program `SortList` which calls a method, `quickSort` in another class, Sorts. If both classes, `SortList` and `Sorts` were converted, then each statement in the `quickSort` method would be translated into nodes in a $\mathcal{CG}$. Many of these statements would be simple comparisons and swaps. It would take longer to schedule the `invoke` nodes representing these operations then it would to carry them out. By compiling the `Sorts` class into a Java class, and converting the main program into a $\mathcal{CG}$, a more useful program is obtained. A call to the `quickSort` method is represented as a single `invoke` node, which when fired executes the `quickSort` method. This method may involve the execution of many Java instructions though it is a single node. Thus the grain size is much coarser than if we compiled all classes.

Better results are obtained by keeping related code together. For example, it would be inefficient to spread nodes controlling a loop across different machines, since they will access each other often. To prevent this, code blocks, such as methods and loop bodies are converted to seperate $\mathcal{CG}$s rather than nodes in a single $\mathcal{CG}$. This facilitates easy distribution of code to machines where it can be unpacked and executed.

5.4. Preprocessing. Before translation begins, the Java code is altered to remove any primitives, and primitive operations, and all arrays are replaced with equivalent calls to the Java Array class. The effect of this is that all variables will be true objects, that is, subclasses of Object, and all actions upon these objects will be carried out via method invocations on those objects. This step is vital since primitive operations are not supported in dynamic invocation in Java.

The first step, removing all primitives, is trivial. Primitive variable declarations e.g., `int num = 7` are rewritten using the relevant wrapper type e.g., `Integer num = new Integer(7)`. This has no effect on methods which require an int but are now passed an Integer, since the Java dynamic invocation mechanism automatically handles this conversion.

All operations on these primitives e.g. `-, ++, <` are converted to method calls to a class written specifically for this purpose. For example, `num < limit` becomes `Operators.lessThan(num, limit)`. The methods in the Operators class are overloaded to handle all primitive types.

The Java Array class is used in place of standard array access. To look up an element in an array, e.g., `list[n]`, a call to `Array.get` is used e.g., `Array.get(list, n)`.

After these modifications are carried out, the code is still valid Java and will execute in the same manner as the original.

Several translation rules are applied to this code to produce an equivalent $\mathcal{CG}$ application.

6. Translation rules.

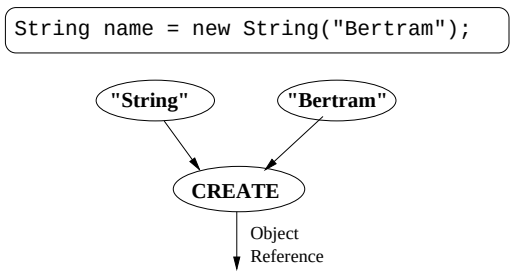


FIG. 6.1. Corresponding Graph for Object Creation

6.1. Object Creation. In $\mathcal{CG}$s, there are no variable names as there are in Java. Relationships such as parameter passing are expressed by linking together nodes. To convert the Java code for objects creation e.g., `ClassName objName=new ClassName(prm);`
only the class name and any parameters are needed, which are used as inputs to a `create` node, as in (Fig. 6.1). The variable `objName` is now associated with this `create` node, and any reference to this variable will be replaced with a reference to this node.

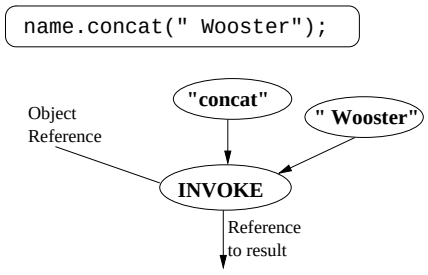


FIG. 6.2. Graph for invocation of a method on an object

6.2. Method Invocation. To invoke a method on an object we need a reference to the object, a method name and, optionally, parameters. The object reference will be an output from a `create` node or a result from another `invoke`. To convert a piece of code of the form:

```
varName = objName.method(param);
```

an `invoke` node is used. The node previously associated with `objName` (from a create or invoke) is used as the object reference. The method name and params make up the remaining inputs, as in (Fig. 6.2). `varName` is now associated with this `invoke` node.

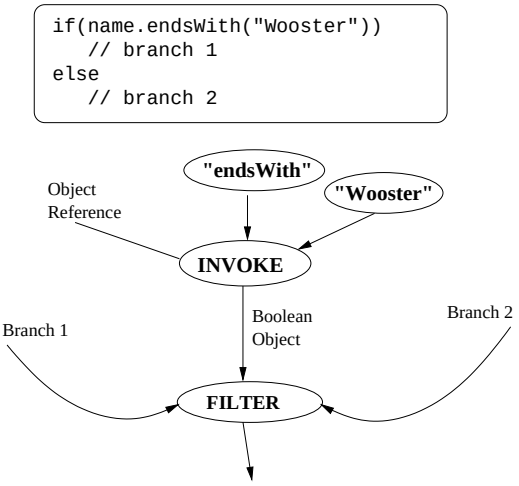


FIG. 6.3. Graph for If-Else statement using Filter node

6.3. If-Else statement. The if-else statement is converted to a Filter node, which takes a boolean parameter and two other nodes, representing the two branches of code to be executed. When the Filter node fires it connects one of the branches to its destination, depending on the truth value of the boolean.

To convert code of the form seen in (Fig. 6.3) a Filter node is added to the graph. If the condition is expressed as a variable e.g., if(property) then the node associated with that variable is added as the boolean input to the Filter node. If the condition, like the branches of code, consists of one or more statements, then those statements are converted to a graph using the translation rules and that graph is used as an input.

The ternary operator is handled identically.

6.4. Loops. In general, with while loops the amount of iterations cannot be calculated at compile time, whereas with for loops it is easily determined. To convert a for loop which results in `n` iterations of some code, we use `spread` node that when fires produces n instances of that code. The `spread` node is a special node that can fire multiple times with one destination.

Caution must be exercised when converting for loops since some are not amenable to this type of parallelisation.

To convert code of the form:

```
for(AnInteger i=0;i.lessThan(100);i.add(1))
    aMethod(i);
```

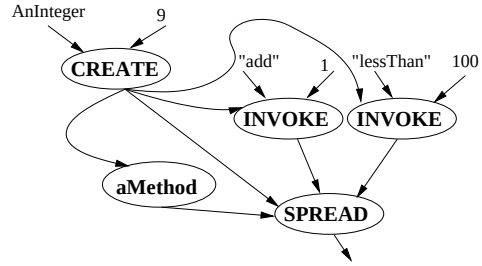


FIG. 6.4. Graph for a For loop

```
for(initCode;conditionCode;incrementorCode)
    mainCode;
```

the initCode, conditionCode and incrementorCode are converted as per the translation rules and added as inputs as in (Fig. 6.4). The initCode can fire immediately since it has a destination. The `spread` node itself can fire too, which causes the conditionCode to execute. If this code returns true then an instance of the mainCode is created. This will continue to happen until the condition becomes false, at which time the `spread` node becomes unfireable. Hence, each time the `spread` node fires, it starts an instance of the mainCode. This is a better approach to starting all N instances at once, since N may be unfeasibly large, or resources may be tight. This method starts as many instances as possible as soon as possible. For a large loop, as the first instances complete execution, more instances can be created as necessary.

An optimised version of the `spread` node takes a mainCode and an integer parameter, N. When fired it creates N instances of the mainCode node and populates each with a unique integer between 0 and N-1.

While loops are handled similarly but only one instance of the code is executed at a time.

```
public String repeat(String str)
{
   return str.concat( str );
}
```

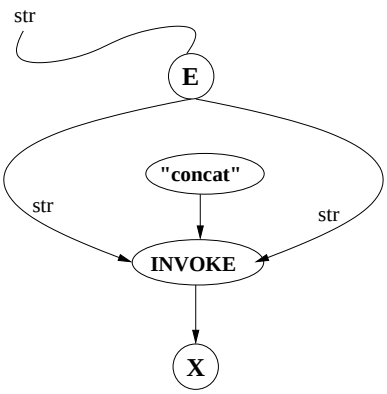


FIG. 6.5. Corresponding Graph for Methods

6.5. Methods. Methods are compiled into $\mathcal{CG}$s in order to keep the related code packaged together for efficient distribution. A $\mathcal{CG}$ has an E (entry) node, which takes any parameters and an X (exit) node, which returns the result.

To convert code of the form:

```
public returnType methodName(params)
{   // code;
```

```
    return result;
}
```

E node is used which takes the supplied parameters as inputs. The method code is converted using the translation steps, with the E node linked to any portions of code which use a parameter. The node associated with the variable being returned is linked to an X node, as in (Fig. 6.5) This completed $\mathcal{CG}$ is given the same name as the Java method including parameter information to prevent ambiguity in cases of method overloading.

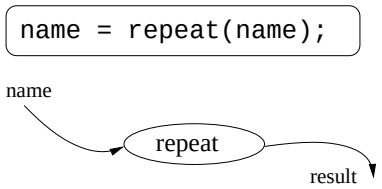


FIG. 6.6. Corresponding Graph for calling a method

To invoke a local method, a node representing the relevant $\mathcal{CG}$ is used, populated with any necessary parameters, as in (Fig. 6.6).

Note the lack of `invoke` node, since this is not an actual method invocation, but a call to another $\mathcal{CG}$.

6.6. Code blocks. Code between curly brackets e.g., {, }, are treated as related code like methods, and are converted into $\mathcal{CG}$s. Any variables accessed in the block are treated as variables and are passed in. The code block is replaced with a call to the $\mathcal{CG}$. This facilitates easy distribution of code e.g., in if statements the two branches could be on seperate machines, but related code is still kept together.

6.7. Synchronising Instructions. Instructions that return values and/or take parameters can be naturally organised into a parallel graph, as in (Fig. 6.7) . In this case it's clear which actions should be taken before other actions can be carried out and so parallelism can be uncovered with no possiblity of executing instruction in the wrong order.

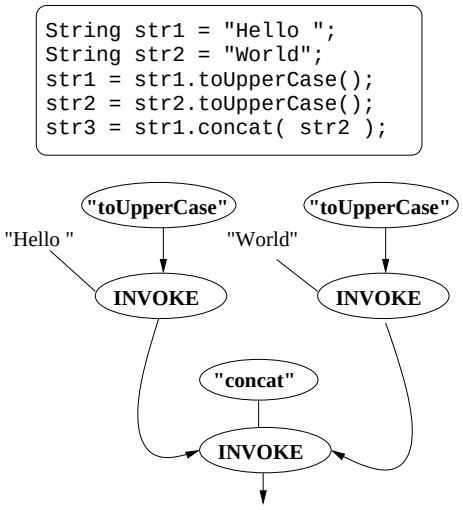


FIG. 6.7. Uncovering Hidden Parallelism

For instructions that return no value, we must ensure instructions are performed in the order the programmer states. To convert code of the form:

```
statement1();
statement2();
```

Converting this code results in two unconnected `invoke` nodes. They must be connected in a manner which guarantees the correct order of execution.

We introduce a `sync` node which takes two parameters corresponding to the statements to be ordered. In (Fig. 6.8) the second statement has no destination so cannot fire. The first statement can fire, and upon

completion a null result is sent to the `sync` node. This fires the `sync` node, which connects it's second parameter, the other statement, to its destination allowing it to fire. This procedure can be used repreatedly to enforce the execution order of any number of instructions.

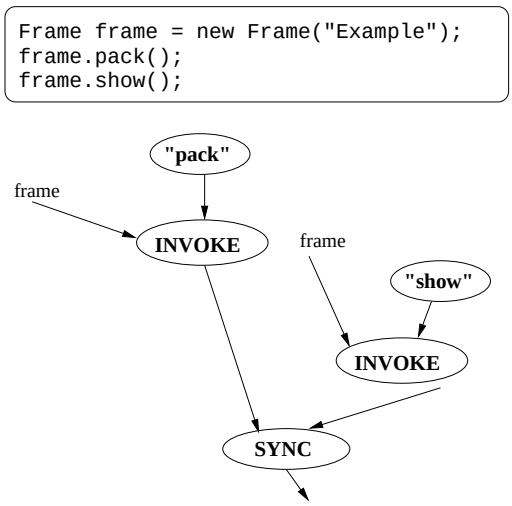


FIG. 6.8. Forcing sequential execution

These translations are enough to enable us to convert most Java programs into $\mathcal{CG}$s.

7. Adding DCOM and CORBA support. Two extra commands, `newDCOM` and `newCORBA`, have been added to this Java grammar to enable support for DCOM and CORBA objects. These are converted just like the Java `new` operator, and method invocations are written just like standard Java invocations. At conversion time, a `create` or `createDCOM` node is produced, depending on the keyword. This also affects the type of invoke node generated later on. Adding this support gives us a very powerful tool that lets us incorporate DCOM and CORBA objects into programs as if they were Java objects.

An example of using a DCOM object called MathsLibrary to test primality of a Java int.

```
DCOM mathsLib = newDCOM MathsLibrary();
int y = 777;
boolean prime = mathsLib.isPrime( y ); // invocation on a DCOM object, assigned
                                       // to a Java primitive
```

A more complex example could interrogate a CORBA object for parameters, use these to create a computation which can be executed across the network and plot the results in, for example, Microsoft Excel.

This method allows a programmer to incorporate any of the many industry applications which supply DCOM or CORBA interfaces into their application. However, since these keywords are not part of the Java language, programs using these features will not compile using a Java compiler. They can only be executed using this system.

8. Sample Applications. The following code fragment produces the graph in (Fig. 8.1). The code is simplified for the purposes of this paper, for example, where print() is used a call to System.out.println should be made, however this would result in the production of create and invoke nodes in the resultant graph. This level of complexity is left to the second example.

```
class CheckSum
{
    void checkNumber(BigInteger n)
    {   BigInteger two = new BigInteger("2");
        print(GetSeqLen(n, 0));
        checkNumber(n.add(two));
    }

    int GetSeqLen(BigInteger m, int i)
```

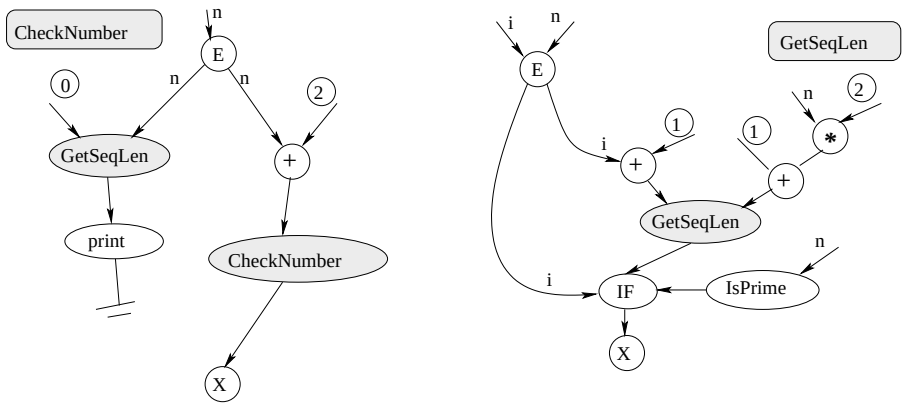


FIG. 8.1. A graph to print the length of sequences of primes generated by taking a prime, doubling it and adding one e.g., 89, 179, 359, 719. The graph is simplified for brevity; isPrime represents an invocation of `isProbablePrime` on the BigInteger N. * and + correspond to the methods `multiply` and `add`. The shaded nodes represent Condensed Graph nodes

```
    {   BigInteger one = new BigInteger("1");

        if(m.isProbablePrime(1000000))
            return GetSeqLen(m.add(m).add(one), i+1);
        else
                return i;
    }
}
```

The example graph in (Fig. 8.1) is built using only standard Java classes. It is a simple recursive graph, used to determine the lengths of sequences of numbers satisfying a certain property. The grain size for this application is too fine to experience a speed up, and actually slows down if run across computers. It is only useful to distribute multiply and addition instructions when the numbers involved are enormous. The application is provided to illustrate the concepts involved in building a $\mathcal{CG}$ from Java objects.

A more realistic example would use at least some custom written code. The application in (Fig. 8.2) uses a class, `MandelLib`, to facilitate calculation of the Mandelbrot Set. The class contains a method `CalcRow` which takes a parameter representing the row to be calculated and returns that row. Parameters such as width, height and magnification are specified at run-time. The graph uses a method, `PlotRow` in another class, `CGWindow` to plot these rows. CGWindow is a general purpose class, used for plotting any data. Rows, rather than individual pixels, are calculated and plotted to increase the grain size of the computation. The application is parallelised using a special node `Spread`, which takes an integer parameter, *I*, and a node *Work* and produces *I* copies of the *Work* node, passing each copy a unique number between 0 and *I-1*. Even though the custom written class, `MandelLib`, contains no network code, the application can still be distributed across the network. The Compeer system handles the transfer of work and results, relieving the programmer of that burden.

An interactive Java Mandelbrot viewer was built using a modified version of this application which returns the data generated rather than displaying it. The viewer loads the KDLang definition of the Mandelbrot graph from a file and replaces three parameters, x, y and magnification, in the graph before submitting it to Compeer. The result is displayed on screen. A user can click on any part of the image causing new coordinates to be generated, and a new job to be submitted to Compeer and subsequently displayed.

This illustrates the ease with which certain types of distributed application can be designed using this technique. Similarly, physical fields, non-linear equations and other operations can be calculated across a network and viewed.

Using DCOM or CORBA these results could be integrated directly into other applications e.g., an MS Access database from which reports could later be extracted.

9. Conclusion and Future Directions. The topics of metacomputing, and decentralised computing, have been the focus of much attention in the recent past. The importance of the field is becoming evident in a variety of application domains. Specific directions include Resilient Overlay Networks [1] and the increase in

```
public void doRow(int row, CGWindow window)
{   MandelLib lib = new MandelLib();
    window.plotRow( lib.calcRow( row ) );
}

public static void main(String [] args)
{   CGWindow window = new CGWindow();

    for(int i=0;i<100;i++)
        doRow( i, window );
}
```

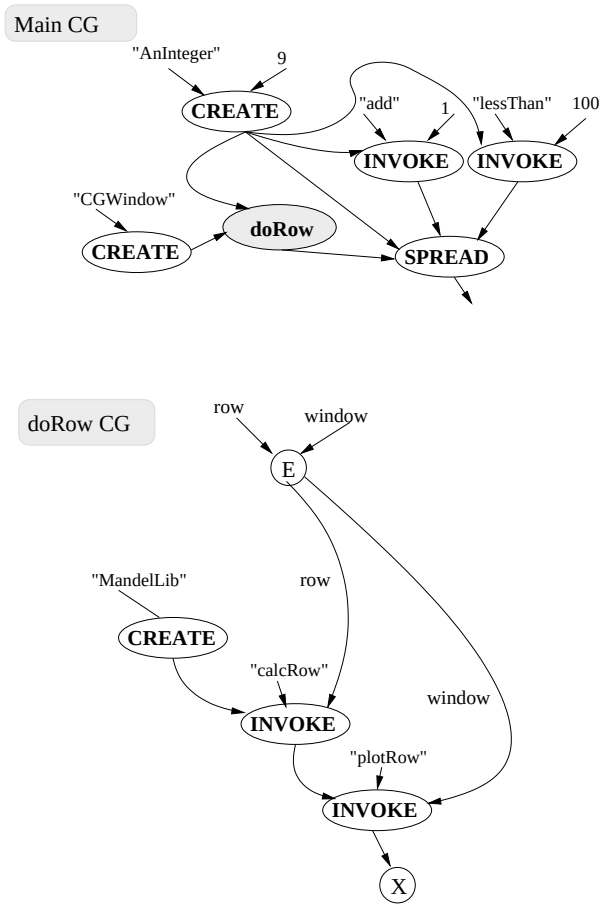


FIG. 8.2. Example program snippet to calculate and display the Mandelbrot Set converted to an automatically parallel $\mathcal{CG}$ The shaded nodes represent Condensed Graph nodes

grid based projects such as Globus.

This paper addresses issues of usability and accessibility. The approach is to use a familiar high-level language and map it onto a dynamically, easily deployed architecture which can by dynamically constructed to execute the task. Novel features include it's exploitation of the $\mathcal{CG}$ computation model to facilitate dynamic load balancing through a process of reflection and applications whose behaviour can be customised at run time.

While the system, as presented here, has obvious benefits in that it reduces the cost in both setting up a metacomputing platform and developing parallel distributed applications, there is much work to be done. For example, the Compeer platform itself needs to be comprehensively benchmarked and compared with other technologies. Methods for safely uncovering more parallelism during the translation process must be investigated and used to maximize the the benefits of distributing the computation. Also, some deployments, particularly

across the Internet, will not become viable until security is properly addressed.

It is envisaged that when these steps are completed, it will be possible for internet based cooperative metacomputers to be constructed in an ad hoc fashion, allowing users to easily share data and services, and create applications to exploit these.

Acknowledgments. The support of the Informatics Research Initiative of Enterprise Ireland is gratefully acknowledged.

REFERENCES

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, *Resilient overlay networks*, in Symposium on Operating Systems Principles, 2001, pp. 131–145.
[2] F. Berman and R. Wolski, *The apples project: A status report*, 1997.
[3] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, *MPJ: MPI-like message passing for Java*, Concurrency: Practice and Experience, 12 (2000), pp. 1019–1038.
[4] distributed.net. `http://www.distributed.net` 2002.
[5] I. Foster and C. Kesselman, *Globus: A metacomputing infrastructure toolkit*, The International Journal of Supercomputer Applications and High Performance Computing, 11 (1997), pp. 115–128.
[6] J. Gosling et al., *The Java Language Specification*, GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan; Unit 1905, Metro Plaza Tower 2, No. 223 Hing Fong Road, Kwai Chung, N.T., Hong Kong, 19xx.
[7] R. Henderson and D. Tweten, *Portable batch system: External reference specification*, tech. report, NASA Ames Research Center, 1996.
[8] JavaSoft, *Java remote method invocation—distributed computing for Java*, white paper, Sun Microsystems, Inc., 1998.
[9] R. Karp and R. Miller, *Properties of a model for parallel computations: Determinacy, termination, queuing.*, SIAM Journal of Applied Mathematics, 14 (1966), pp. 1390–1411.
[10] B. C. Kuszmaul, *Suif to dataflow*. Second SUIF Compiler Workshop, Stanford University, August 1997.
[11] M. S. Lam and R. P. Wilson, *Limits of control flow on parallelism*, in Nineteenth International Symposium on Computer Architecture, Gold Coast, Australia, 1992, ACM and IEEE Computer Society, pp. 46–57.
[12] P. Launay and J.-L. Pazat, *A framework for parallel programming in JAVA*, in HPCN Europe, 1998, pp. 628–637.
[13] M. J. Lewis and A. Grimshaw, *The core legion object model*, Tech. Report CS-95-35, 1995.
[14] D. A. P. . J. P. Morrison, *Nectere: A general purpose environment for computing on clusters, grids and the internet*, in International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '02), Las Vegas, USA, June 2002, pp. 719–726.
[15] T. Parr and R. Quong, *Antlr: A predicatedll (k) parser generator*, Journal of Software Practice and Experience, 25 (1995), pp. 789–810.
[16] M. Philippsen and M. Zenger, *JavaParty — transparent remote objects in Java*, Concurrency: Practice and Experience, 9 (1997), pp. 1225–1242.
[17] L. F. Sarmenta, *Sabotage-tolerance mechanisms for volunteer computing systems*, in 1st International Symposium on Cluster Computing and the Grid, Brisbane, Australia, May 2001, p. 337.
[18] B. Schneier, *Applied Cryptography*, John Wiley And Sons, 2nd ed., 1996.
[19] SETI@Home. http://setiathome.ssl.berkerly.edu, 2002.
[20] A. Silis and K. A. Hawick, *The DISCWorld Peer-To-Peer Architecture*, in Proc. of the 5th IDEA Workshop, Fremantle, 1998.
[21] G. software GmbH. `http://www.genias.de/genias/english/codine.html` September 1995.
[22] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su, *Running EveryWare on the computational grid*, 1999.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: October 01, 2002
Accepted: December 15, 2002

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 33–43. http://www.scpe.org
ISSN 1895-1767

THE ROLE OF XML WITHIN THE WEBCOM METACOMPUTING PLATFORM

JOHN P. MORRISON*, PHILIP D. HEALY*, DAVID A. POWER* AND KEITH J. POWER*

Abstract. Implementation details of the Nectere distributed computing platform are presented, focusing in particular on the benefits gained through the use of XML for exoressing, executing and pickling computations. The operation of various Nectere features implemented with the aid of XML are examined, including communication between Nectere servers, specifying computations, code distribution, and exception handling. The area of interoperability with common middleware protocols is also explored.

Key words. Nectere, WebCom, XML, Distributed Computing, Condensed Graphs

1. Introduction. The development of distributed applications has been greatly simplified in recent years by the emergence of a variety of standards, tools and platforms. Traditional solutions such as RPC [1] and DCE [2] have been joined by a variety of new middleware standards and architecture independent development platforms. CORBA [3], a middleware standard developed by the Object Management Group, has seen widespread acceptance as a means for constructing distributed object-oriented applications in heterogeneous environments, using a variety of programming languages. Microsoft's DCOM [4] fulfills a similar role in Windows environments. Sun Microsystems' RMI [5] has been developed as a means of creating distributed applications with the Java platform. The Java platform itself facilitates development for heterogeneous environments due to its architectural independence. Microsoft have recently launched a similar, competing platform in the form of their .NET initiative [6].

Although the various platforms and middlewares described above facilitate distributed application development, their use of binary protocols hinders interoperability and the ability of humans to understand the data being communicated. Because of these limitations, such middlewares are often unsuitable for applications distributed over the Internet, such as Web Services and Business to Business applications [7]. These factors, along with the realization that proprietary binary protocols can lead to "vendor lock-in", have led to a demand for an open, standardized, text-based format for exchanging data. XML [8], a restricted form of SGML developed by the World Wide Web Consortium (W3C), fills this role. XML simplifies the task of representing structured data in a clear, easily understandable (by both machines and humans) format. Furthermore, Document Type Definitions (DTDs) and XML Schemas can be used to enforce constraints on the structure and content of XML documents.

XML forms the basis for several distributed computing protocols and frameworks. SOAP (Simple Object Access Protocol) [9], developed by the W3C is a lightweight protocol for exchange of information in a decentralized, distributed environment. XML-RPC [10] is a simple remote procedure calling protocol. Microsoft's BizTalk Framework [11] provides an XML framework for application integration and electronic commerce. Sun have provided a rich framework for the construction of Java/XML based distributed applications, including XML processing (JAXP), XML binding for Java objects (JAXB), XML messaging (JAXM), XML registries (JAXR) and remote procedure calls with XML (JAX-RPC) [12]. These technologies, due to their open, text-based nature, can utilize existing Internet protocols such as HTTP and SMTP, bypassing firewalls and enabling Internet based applications.

The field of *Metacomputing*, defined as *"the use of powerful computing resources transparently available to the user via a networked environment"* [13], has attracted considerable interest as a means of further simplifying distributed application development. Although this definition is quite vague, in practice metacomputing has come the mean the use of middlewares to present a collection of potentially diverse and geographically distributed computing resources transparently to the user as a single virtual computer. Examples of metacomputing environments are the Globus Metacomputing Toolkit [14], Legion [15] and WebCom (the focus of this paper). WebCom is a metacomputing environment that allows computations expressed as *Condensed Graphs* (see Section 2) to be executed on a variety of platforms in a secure, fault-tolerent manner. Load-balancing is also performed over the computing resources available without requiring any intervention on the part of the programmer. Originally designed as a means of creating *ad hoc* metacomputers from Java applets embedded in web pages, WebCom has since been developed into a general-purpose distributed computing environment suitable for the creation of grids. An extended version of WebCom, entitled WebCom-G [16], allows for interoperability with other Grid Computing platforms and includes support for legacy applications.

*Computer Science Dept., University College Cork, Ireland.

The organization of the remainder of this paper is as follows: Section 2 describes the Condensed Graphs model of computation that provides the underlying execution model for WebCom. Section 3 provides an overview of the design and operation of the WebCom metacomputer itself. The XML file format developed for specifying WebCom applications is presented in Section 4. The pickling of live WebCom computations is discussed in Section 5. A web service interface to WebCom is presented in Section 6. Some empirical results pertaining to the bandwidth savings achievable through the use of compressed XML compared to compressed serialized Java objects are provided in Section 7. Finally, conclusions and a discussion on future work is presented in Section 8.

2. Condensed Graphs. While being conceptually as simple as classical data-flow schemes [17, 18], the Condensed Graphs ($\mathcal{CG}$) model [19] is far more general and powerful. It can be described concisely, although not completely, by comparison. Classical dataflow is based on data dependency graphs in which nodes represent operators, and edges are data paths conveying simple data values between them. Data arrive at *operand ports* of nodes along input edges and so trigger the execution of the associated operator (in dataflow parlance, they cause the node to *fire*). During execution these data are consumed and a resultant datum is produced on the node's outgoing edges, acting as input to successor nodes. Operand sets are used as the basis of the firing rules in data-driven systems. These rules may be *strict* or *non-strict*. A strict firing rule requires a complete operand set to exist before a node can fire; a non-strict firing rule triggers execution as soon as a specific proper subset of the operand set is formed. The latter rule gives rise to more parallelism but also can result in overhead due to remaining packet garbage (RPG).

Like classical dataflow, the $\mathcal{CG}$ model is graph-based and uses the flow of entities on arcs to trigger execution. In contrast, $\mathcal{CG}$s are directed acyclic graphs in which every node contains, not only operand ports, but also an operator and a destination port. Arcs incident on these respective ports carry other $\mathcal{CG}$s representing operands, operators, and destinations. Condensed Graphs are so called because their nodes may be condensations, or abstractions, of other $\mathcal{CG}$s. (Condensation is a concept used by graph theoreticians for exposing meta-level information from a graph by partitioning its vertex set, defining each subset of the partition to be a node in the condensation, and by connecting those nodes according to a well-defined rule [20].) Condensed Graphs can thus be represented by a single node (called a *condensed node*) in a graph at a higher level of abstraction. The number of possible abstraction levels derivable from a specific graph depends on the number of nodes in that graph and the partitions chosen for each condensation. Each graph in this sequence of condensations represents the same information at a different level of abstraction. It is possible to navigate between these abstraction levels, moving from the specific to the abstract through condensation, and from the abstract to the specific through a complimentary process called *evaporation*.

The basis of the $\mathcal{CG}$ firing rule is the presence of a $\mathcal{CG}$ in every port of a node. That is, a $\mathcal{CG}$ representing an operand is associated with every operand port, an operator $\mathcal{CG}$ with the operator port, and a destination $\mathcal{CG}$ with the destination port. This way, the three essential ingredients of an instruction are brought together (these ingredients are also present in the dataflow model; only there, the operator and destination are statically part of the graph).

A condensed node, a node representing a datum, and a multinode $\mathcal{CG}$ can all be operands. A node represents a datum with the value on the *operator* port of the node. Data are then considered as zero-arity operators. Datum nodes represent graphs which cannot be evaluated further and so are said to be in *normal form*. Condensed node operands represent unevaluated expressions, they cannot be fired since they lack a destination. Similarly, multinode $\mathcal{CG}$ operands represent partially evaluated expressions. The processing of condensed node and multinode operands is discussed below.

Any $\mathcal{CG}$ may represent an operator. It may be a condensed node, a node whose operator port is associated with a machine primitive (or a sequence of machine primitives), or it may be a multinode $\mathcal{CG}$.

The present representation of a destination in the $\mathcal{CG}$ model is as a node whose own destination port is associated with one or more port identifications. The expressiveness of the $\mathcal{CG}$ model can be increased by allowing any $\mathcal{CG}$ to be a destination but this is not considered further here. Fig. 2.1 illustrates the congregation of instruction elements at a node and the resultant rewriting that takes place.

When a $\mathcal{CG}$ is associated with every port of a node it can be fired. Even though the $\mathcal{CG}$ firing rule takes accounts of the presence of operands, operators and destinations, it is conceptually as simple as the dataflow rule. Requiring that the node contain a $\mathcal{CG}$ in every port before firing prevents the production of RPG. As outlined below, this does not preclude the use of non-strict operators or limit parallelism.

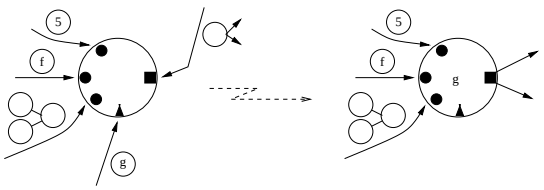


FIG. 2.1. *cgs congregating at a node to form an instruction.*

A *grafting* process is employed to ensure that operands are in the appropriate form for the operator: non-strict operators will readily accept condensed or multinode $\mathcal{CG}$s as input to their non-strict operands. Strict operators require all operands to be data. Operator strictness can be used to determine the strictness of operand ports: a strict port must contain a datum $\mathcal{CG}$ before execution can proceed, a non-strict port may contain any $\mathcal{CG}$. If, by computation, a condensed or multinode $\mathcal{CG}$ attempts to flow to a strict operand port, the *grafting* process intervenes to construct a destination $\mathcal{CG}$ representing that strict port, and sends it to the operand.

The grafting process thus facilitates the evaluation of the operand by supplying it with a destination and, in a well constructed graph, the subsequent evaluation of that operand will result in the production of a $\mathcal{CG}$ in the appropriate form for the operator. The grafting process, in conjunction with port strictness, ensures that operands are only evaluated when needed. An inverse process called *stemming* removed destinations from a node to prevent it from firing.

Strict operands are consumed in an instruction execution but non-strict operands may be either consumed or propagated. The $\mathcal{CG}$ operators can be divided into two categories: those that are "value-transforming" and those that only move $\mathcal{CG}$s from one node to another (in a well-defined manner). Value-transforming operators are intimately connected with the underlying machine and can range from simple arithmetic operations to the invocation of sequential subroutines and may even include specialized operations like matrix multiplication. In contrast, $\mathcal{CG}$ moving instructions are few in number and are architecture independent. Two interesting examples are the condensed node operator and the *filter* node. Filter node have three operand ports: a Boolean, a *then*, and an *else*. Of these, only the Boolean is strict. Depending on the computed value of the Boolean, the node fires to send either the *then* $\mathcal{CG}$ or the *else* $\mathcal{CG}$ to its destination. In the process, the other operand is consumed and disappears from the computation. This action can greatly reduce the amount of work that needs to be performed in a computation if the consumed operands represent an unevaluated or partially evaluated expression. All condensed node operators are non-strict in all operands and fire to propagate all their operands to appropriate destinations in their associated graph. This action may result in condensed node operands (representing unevaluated expressions) being copied to many different parts of the computation. If one of these copies is evaluated by grafting, the graph corresponding to the condensed operand will be invoked to produce a result. This result is held local to the graph and returned in response to the grafting of the other copies. This mechanism is reminiscent of parallel graph reduction [21] but is not restricted to a purely lazy framework.

$\mathcal{CG}$s which evaluate their operands and operator in parallel can easily be constructed by introducing *spec* (speculation) nodes to act as destinations for each operand. The spec node has a single operand port which is strict. The multinode $\mathcal{CG}$ operand containing the spec node is treated by non-strict operand ports in the same way as every other $\mathcal{CG}$, however, if it is associated with a strict port, the spec node's operand is simply transferred to that port. If that operand already had fully evaluated it could be used directly in the strict port, otherwise, it is grafted onto the strict port as described above. This is illustrated in Fig. 2.2.

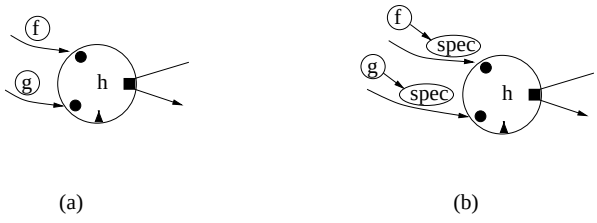


FIG. 2.2. *Increasing parallelism by speculatively evaluating operands.*

Stored structures can be naturally represented in the $\mathcal{CG}$ model. $\mathcal{CG}$s are associated with the operand ports of a node which initially contain no operator or destination. These operands structures can then be fetched by sending appropriate *fetch* operators and destinations to these nodes. These fetch operators also form part of the $\mathcal{CG}$ moving operators and so are machine independent.

The power of the operators in the $\mathcal{CG}$ model can be greatly enhanced by, associating with each, specific *deconstruction semantics*. These specify if $\mathcal{CG}$s can be removed from the ports of a node after firing. In general, every node will be deconstructed to remove its destination after firing, this renders a node incomplete and prevents it from being erroneously refired. The deconstruction semantics of the fetch operator cause the operator and the destination to be removed after firing. This leaves the stored structure in its original state ready for a subsequent fetch. Fig. 2.3 illustrates the process of fetching a stored structure.

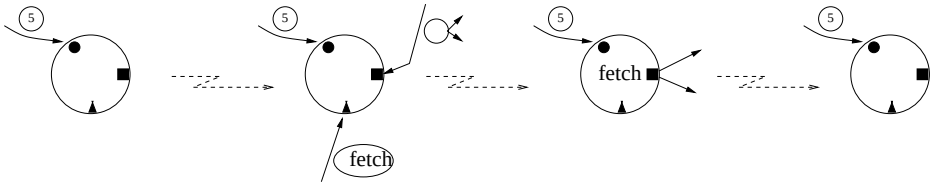


FIG. 2.3. *Sequence of events showing the fetching of a stored structure and subsequent node deconstruction.*

By statically constructing a $\mathcal{CG}$ to contain operators and destinations, the flow of operand $\mathcal{CG}$s sequence the computation in a dataflow manner. Similarly, constructing a $\mathcal{CG}$ to statically contain operands and operators, the flow of destination $\mathcal{CG}$s will drive the computation in a demand-driven manner. Finally, by constructing $\mathcal{CG}$s to statically contain operands and destinations, the flow of operators will result in a control-driven evaluation. This latter evaluation order, in conjunction with side-effects, is used to implement imperative semantics. The power of the $\mathcal{CG}$ models result from being able to exploit all of these evaluation strategies in the same computation, and dynamically move between them, using one single, uniform, formalism.

3. WebCom. Problem solving for parallel systems traditionally lay in the realm of message passing systems such as PVM and MPI on networks of distributed machines, or in the use of specialised variants of programming languages like Fortran and C on distributed shared memory supercomputers. The WebCom System [22, 23, 24, 25] detailed here relates more closely to message passing systems, although it is much more powerful. Message passing architectures normally involve the deployment of a codebase on client machines, and employ a master or server to transmit or *push* "messages" to these clients. WebCom supports this level of communication, but is unique in bootstrapping its codebase by *pulling* it from the server as required.

Technologies such as PVM, MPI and other metacomputing systems place the onus on the developer to implement complete parallel solutions. Such solutions require a vast knowledge on the programmer's part in understanding the problem to be solved, decomposing it into its parallel and sequential constituents, choosing and becoming proficient in a suitable implementation platform, and finally implementing necessary fault tolerance and load balancing/scheduling strategies to successfully complete the parallel application. Even relatively trivial problems tend to give rise to monolithic solutions requiring the process to be repeated for each problem to be solved.

WebCom removes much of these traditional considerations from the application developer; allowing solutions to be developed independently of the physical constraints of the underlying hardware. It achieves this by employing a two level architecture: the computing platform and the development environment. The computing platform is implemented as an Abstract Machine (AM), capable of executing applications expressed as Condensed Graphs. Expressing applications as Condensed Graphs greatly simplifies the design and construction of solutions to parallel problems. The Abstract Machine executes tasks on behalf of the server and returns results over dedicated sockets. The computing platform is responsible for managing the network connections, uncovering and scheduling tasks, maintaining a balanced load across the system and handling faults gracefully. Applications developed with the development environment are executed by the abstract machine. The development environment used is specific for Condensed Graphs. Instructions are typically composed of both sequential programs (also called atomic instructions) and Condensed nodes encapsulating graphs of interacting sequential programs. In effect, a Condensed Graph on WebCom represents a hierarchical job control and specification language. The same Condensed Graphs programs execute without change on a range of implemen-

tation platforms from silicon based Field Programmable Gate Arrays[26] to the WebCom metacomputer and the Grid.

Imposing a clear separation between the abstract machine and the development environment has many advantages. Fault tolerance, load balancing and the exploitation of available parallelism are all handled implicitly by WebCom abstract machines without the need for programmer intervention. Also, different strategies such as load balancing strategies can be used without having to redesign and recode any applications that may execute on the system. Removing the necessity for developers to implement the features mentioned previously, which can be considered as core to any distributed computing system, greatly simplifies and increases the rate of development of parallel applications.

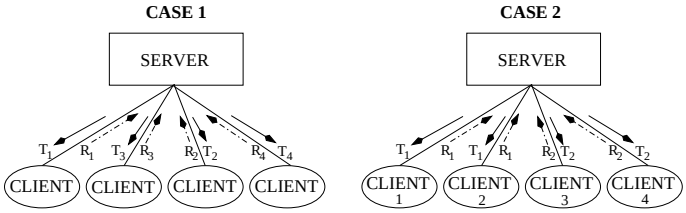


FIG. 3.1. *Traditional Server/Client Metacomputers consist of a two tier hierarchy. The server sends tasks to the clients.*

Physical compute nodes connect over a network to form a hierarchy. Metacomputing systems typically consist of one server (or master) and a number of clients. Tasks are communicated from the server to the clients, and results returned when task computation is completed. This topology represents a two tier unidirectional connectivity hierarchy. Tasks are only transmitted in one direction from the server to the clients and results of task execution transmitted back, as illustrated in Figure 3.1. Case 1 is reminiscent of the *"...@home"* projects like seti@home and genome@home. With these projects, a generic task is installed apriori on each computer by its local administrator. These tasks register with a server which subsequently supplies input data and gathers results on an ongoing basis. Fault tolerance is achieved by issuing the same input data to multiple tasks.

Case 2 depicts a more sophisticated arrangement. Clients in this architecture represent codebases which may be invoked with specific parameters by the server. Each client may comprise a different codebase, thereby requiring the server to target instructions. These systems typically invoke the same task on multiple available clients in an effort to minimise response time: the earliest result received for each task is used and all others are discarded.

Two tier systems have limited scalability, particularly when distributing fine grained tasks: the server will eventually become inundated with client connections and will expend a disproportionate effort in connection management. The seti@home project alleviates this problem by allowing clients to disconnect after receiving substantial data blocks that can take days to process. An advantage of two tier hierarchies is simple fault tolerance and load balancing strategies can be easily implemented.

WebCom also utilises a server/client model and World Wide Web technologies. In contrast, A WebCom client is an *Abstract Machine*(AM) whose codebase is populated on demand. In certain instances client machines may act as servers capable of distributing tasks to clients of its own. This feature gives rise to a multi-tier bidirectional topology. When employing the Condensed Graphs model of computation, the evolution of this topology reflects the unfolding of the Condensed Graph description of the application. An example WebCom connection topology is illustrated in Figure 3.2.

In certain circumstances a client may request its server to execute one or more tasks on its behalf. This mechanism allows maximum exploitation of available compute resources and gives rise to the evolution of peer to peer connectivity. Within a WebCom deployment, peer to peer connections may be established within sections of the hierarchy, resulting in the creation of a moderated peer to peer network. For deployments consisting of a small number of clients, a fully connected peer to peer topology may evolve. An example evolution of a moderated peer to peer network hierarchy is illustrated in Figure 3.3.

A deployment of WebCom consists of a number of AMs. When a computation is initiated one AM acts as the root server and the other AMs act as clients or *Promotable Clients*. The promotability of a client is determined by the deployment infrastructure. A promoted client acting as a server accepts incoming connections and executes tasks. Promotion[27, 28] occurs when the task passed to the client can be partitioned for subsequent distributed execution (i.e., when the task represents a Condensed Graph). It is possible for the server to be redirected or to act as a client of a promotable client. This situation normally results in the evolution of a peer

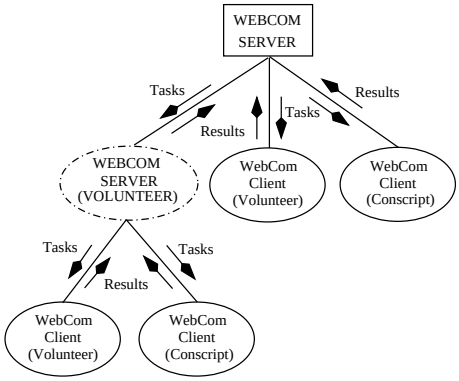


FIG. 3.2. *Typical WebCom Client/Server connectivity. Each WebCom Abstract Machine can either volunteer cycles to a WebCom server, or be conscripted as a WebCom client through Cyclone.*

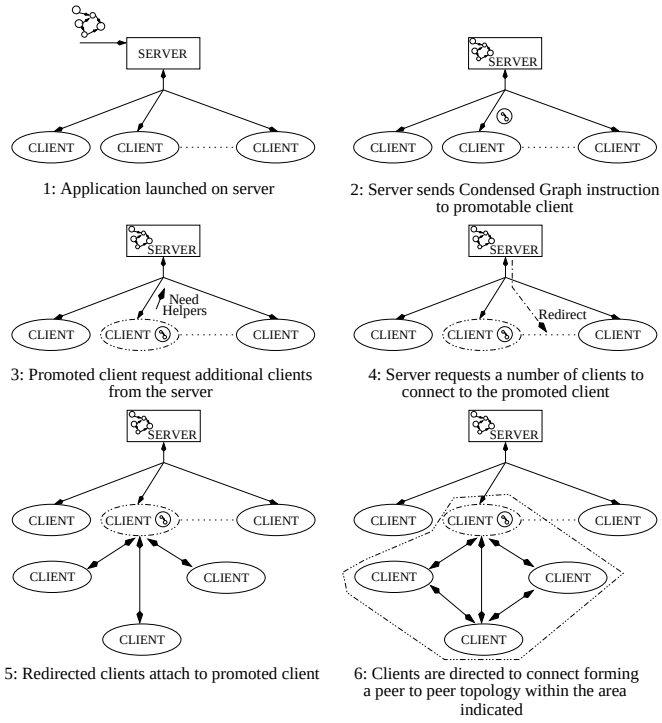


FIG. 3.3. *Evolution of a Moderated Peer to Peer network. 1: Computation begins with 2 tier hierarchy. Graph execution begins on the server. 2: Server sends a Condensed Graph task to a client, causing it to be promoted. 3. The promoted client requests additional clients to be assigned to it by the server. This request may not be serviced. If not, the client continues executing the graph on its own. 4: The server issues a redirect directive to a number of its clients. 5: Clients redirected from the server connect as clients of the promoted client. 6: The promoted client directs its new clients to connect to each other forming a local peer to peer network.*

to peer topology. The return of a result may trigger the demotion of a promoted client, thus causing it to act once more as a normal client.

Furthermore, WebCom clients are uniquely comprised of both volunteers and conscripts. Volunteers donate compute cycles by instantiating a web based connection to a WebCom server. The AM, constrained to run in the browser's sandbox, is automatically downloaded and establishes communications with the server. Due to the constraints associated with executing within the sandbox, such as prohibiting inbound connections, volunteer clients are not promotable: they will only be sent primitive tasks for execution. Clients that have the WebCom AM pre-installed may also act as volunteers. These clients are promotable, as they are not constrained by the restrictions imposed by executing within a browser.

Intranet clients must be conscripted to WebCom by the Cyclone[29] server. Cyclone directs machines under its control to install the WebCom AM locally thereby causing them to attach to a predefined server and to act as promotable clients. These clients may be further redirected by that server.

The Abstract Machine(AM) consists of a number of modules including a Backplane and modules for Communications, Fault Tolerance, Load Balancing and Security. For better tuning to specific application areas, the AM allows every component to be dynamically interchangeable. The pluggable nature of each WebCom module provides great flexibility in the development and testing of new components. Certain components maybe necessary, while others could be highly specialised to particular application environments. A skeletal installation is composed of a Backplane module, a Communication Manager Module and a number of module stubs as illustrated in Figure 3.4.

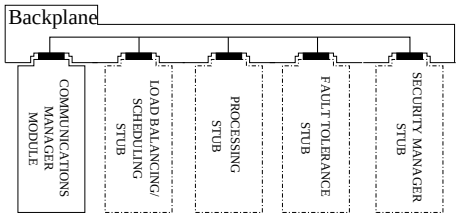


FIG. 3.4. *A minimal WebCom installation consists of a Backplane Module, a Communication Manager Module and a number of module stubs for Processing, Fault Tolerance, Load Balancing and Security.*

The Backplane acts as an interconnect and bus that co-ordinates the activities of the modules via a well defined interface. Inter module communication is carried out by placing a task on the Backplane. The Backplane interrogates the task to determine whether it should be routed to a local module or to another AM via the communications manager. This mechanism provides great flexibility especially as the mechanism can be applied transparently across the network: transforming the metacomputer into a collection of networked modules. This allows an arbitrary module to request information from local modules or modules installed on different abstract machines. For example, when making load balancing decisions, a server's Load Balancing Module can request information from the Load Balancing Module of each of its clients. A high level view of multiple AM's on a network is given in Figure 3.5

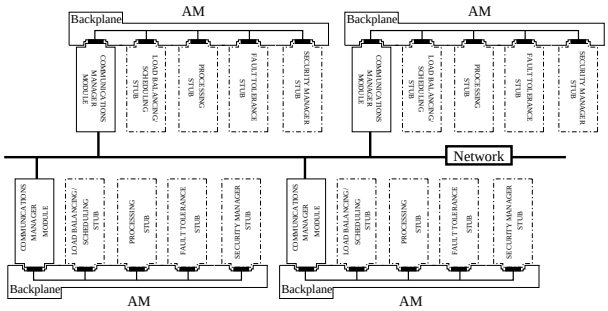


FIG. 3.5. *High level view of abstract machines connected over a network.*

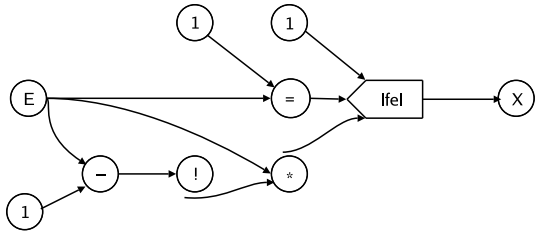
Inter module and inter AM communication is carried out by the Backplane. A *message* representing a task, lies at the core of the communication structure. To summarise, when a module initiates communication with another local or remote module, a message is created specifying the task to be executed along with certain execution environment parameters. The message is placed on the Backplane which interrogates it to determine suitability for distribution. If the message is to be executed remotely it is passed to the Connection Manager Module for subsequent dispatch, otherwise it is passed to the appropriate local module. Upon receipt of a message object, a module immediately processes the associated task.

4. Graph Definition XML Format. Early versions of WebCom required the implementation of a Java class for every graph definition used by an application. As development progressed, it became clear that this approach suffered from a number of significant limitations, such as requiring knowledge of Java and an in-depth understanding of the opearation of the Condensed Graphs model on the part of the application developer. In

order to simplify application development, an IDE was developed that allows Condensed Graph definitions to be expressed in a graphical fashion. Initially, the IDE maintained an internal representation of graph definitions that could be converted to Java code and subsequently compiled. However, applications other than the IDE that wished to generate graph definitions could not easily do so.

It became clear that a common method of expressing graph definitions was required that could be utilized directly both by humans and a variety of potential applications using a variety languages and APIs. XML was the obvious choice for this task due its text-based nature, platform independence and the widespread availability of XML tools across practically every platform. XML also offered the advantage of self-validation using DTDs and XML Schema.

Files written according to the XML format are composed of a a number of graph definitions, represented by `graphdef` elements enclosed in a root `definitions` element. Each `graphdef` element is in turn composed of a number `node` elements that specify the operand ports, operands, operator and destinations that comprise the node. A namespace (with `cg` used as the prefix) is used to allow graph definitions to be embedded in other XML documents and *vice versa*. An example graph and its equivalent XML representation is given in Figure 4.1. Live graph instances can be converted to the XML format via pickling (see Section 5, below).




```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE cg:graphdefs SYSTEM "http://cuc.ucc.ie/xml/cg.dtd">

<cg:graphdefs xmlns:cg="http://cuc.ucc.ie/xml/cg">
  <cg:graphdef name="Factorial">
    <!-- E -->
    <cg:node name="E">
      <cg:operandport/>

      <cg:operatorport operator="webcom.cgengine.EnterOperator"/>

      <cg:destinationport>
        <cg:destination nodename="Eq0" portnumber="0"/>
        <cg:destination nodename="minus1" portnumber="0"/>
        <cg:destination nodename="mul" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- minus1 -->
    <cg:node name="minus1">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="strict"/>

      <cg:operand portnumber="1" type="Integer" value="1"/>

      <cg:operatorport operator="webcom.nodes.core.MinusOp"/>

      <cg:destinationport>
        <cg:destination nodename="Factorial" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- factorial -->
    <cg:node name="Factorial">
      <cg:operandport strictness="strict"/>

      <cg:operatorport operator="Factorial"/>

      <cg:destinationport/>
    </cg:node>

    <!-- mul -->
    <cg:node name="mul">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="strict"/>

      <cg:operand portnumber="1" type="Node" value="Factorial"/>

      <cg:operatorport operator="webcom.nodes.core.MultiplyOp"/>

      <cg:destinationport/>
    </cg:node>

    <!-- Eq0 -->
    <cg:node name="Eq0">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="strict"/>

      <cg:operand portnumber="1" type="Integer" value="0"/>

      <cg:operatorport operator="webcom.nodes.core.EqualsOp"/>

      <cg:destinationport>
        <cg:destination nodename="ifel" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- ifel -->
    <cg:node name="ifel">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="nonstrict"/>
      <cg:operandport strictness="nonstrict"/>

      <cg:operand portnumber="1" type="Integer" value="1"/>

      <cg:operand portnumber="2" type="Node" value="mul"/>

      <cg:operatorport operator="webcom.nodes.core.IfElOp"/>

      <cg:destinationport>
        <cg:destination nodename="X" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- X -->
    <cg:node name="X">
      <cg:operandport strictness="strict"/>

      <cg:operatorport operator="webcom.cgengine.ExitOperator"/>

      <cg:destinationport/>
    </cg:node>
  </cg:graphdef>
</cg:graphdefs>
```

FIG. 4.1. *A trivial example of a graph definition that calculates the factorial of its single argument, along with the equivalent XML document.*

5. Pickling Computations. Pickling is the process of creating a serialized representation of objects [30]. In this case, the objects in question comprise an executing WebCom computation, and the serialized representation will consist of an XML document. Essentially, this allows the state of a computation to be preserved. The computation could be, for example, stored in a database and reactivated at a later date (a feature that could prove useful during the execution of computations that block for long periods of time). Alternatively, the computation could be transported to another environment and executed there, if the necessary resources were available.

In order to reconstruct the state of a WebCom computation, two pieces of information are needed: the V-Graph and the set of $\mathcal{CG}$ definitions from which the computation was created. The V-Graph represents the current state of the computation. The graph definitions are necessary to progress the computation; if a condensed node is executed, it cannot be evaporated unless its definition is available. These pieces of information are usually not contained in any single location; rather they are distributed throughout the various WebCom servers partaking in the computation and must be reconstructed.

Before a computation can be pickled, it must be frozen. This can be partially achieved by instructing the compute engines participating in the computation to cease scheduling new instructions. The situation is complicated by the problem of deciding what to do with partially executed instructions, particularly instructions that may take indefinite periods of time to complete execution. To allow for this problem, three types of halting messages may be sent to the compute engines: The first instructs the compute engines to report their state immediately and to terminate any currently executing instructions. The second instructs the engines to wait indefinitely for all instructions to finish execution. The third allows for a timeout period to be specified before instructions are terminated prematurely.

Once all the participating clients have responded, a complete representation of the computation is now available at the root server. The V-Graph and definition graphs are then converted to an XML document. Since the V-Graph is itself a $\mathcal{CG}$ it is stored according to the standard $\mathcal{CG}$ definition schema, as are the definition graphs.

6. Exposing WebCom as a Web Service. In order to facilitate the utilization of the WebCom platform across a variety of platforms and programming languages, it was decided to expose WebCom as a web service. In effect, this implied making one or more methods remotely available using SOAP and creating descriptions of these methods using WSDL. Due to the text-based nature of SOAP and the ability to use it over common protocols such as HTTP and SMTP, web services can be accessed in a platform-agnostic way, eliminating the need to reimplement APIs on different platforms and languages.

For simplicity, only two methods were exposed: `runGraph` and `runXmlJob`. `runGraph` accepts two parameters (a graph name and a parameter list) and executes the specified graph definition with the supplied parameters. The return value of the graph is returned if the computation finishes succesfully; otherwise an appropriate error message is returned. `runXmlJob` behaves identically to `runGraph` except that it accepts a graph definition document as an additional parameter, with the graph definitions contained in the document added to the computation's definition collection before execution commences. The newly-added definitions are propagated from the root WebCom instance to clients as required via the messaging system.

The Glue web service platform [31] was used to create and deploy the web service. This platform was chosen both for ease of development and deployment. Development is simplified through the use of reflection by the platform to determine the names and types of the methods to be exposed, allowing the web service to be defined with a minimum of coding effort. The corresponding WSDL (Web Services Description Language) descriptions are created automatically, eliminating the need to specify them by hand. Deployment is simplified by the fact that Glue is a standalone platform; there is no requirement for WebCom administrators to install and configure a separate web services container, a requirement of alternative web services platforms such as Apache Axis.

7. Empirical Data on XML Document Sizes. Given the verbosity of XML documents, some empirical data was gathered to determine the overhead imposed by representing $\mathcal{CG}$s as XML. Linear $\mathcal{CG}$s containing 1 to 1,000 nodes were specified as XML documents and loaded into a WebCom server before being converted to their serialized Java equivalents. This allowed the relative sizes of both representations to be compared. Since it would be possible to compress large graphs (such as those created by the pickling process) before transmission, the relative sizes after compression were also compared. The compression schemes tested were JAR, the standard Java archiving tool, and XMill [32], a specialized XML compression utility. Both the serialized Java and XML

representations were compressed with JAR, whereas XMill applies only to XML. The number of operands and destinations in the linear $\mathcal{CG}$ nodes were varied so that nodes were not uniform and hence more easily compressed.

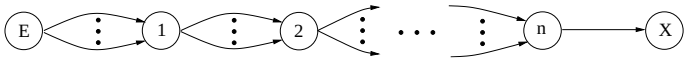


FIG. 7.1. *Structure of the $\mathcal{CG}$s used in the empirical study.*

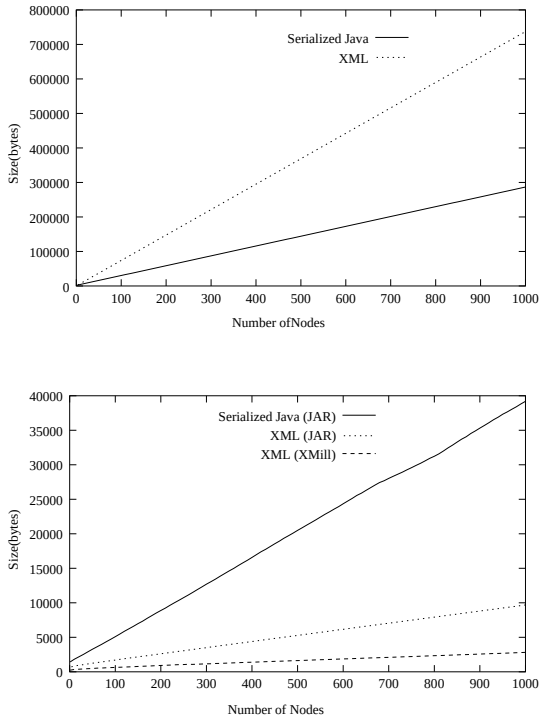


FIG. 7.2. *Graphs showing the relative sizes of $\mathcal{CG}$s represented and compressed using different formats.*

The results of studying 100 graphs with node sizes ranging from 1 to 1000 are illustrated in Fig. 7.2. As expected, graphs represented as XML documents are larger than their serialized Java equivalents, typically by a factor of two. When compressed, however, the XML documents occupy far less space, roughly 1% of their original size for large files, an order of magnitude smaller than the compressed serialized Java files representing the same graphs. These results would indicate that significant storage and bandwidth savings are possible through the utilization of XML compression, at the cost of human readability. A possible application of these results would be to compress extremely large $\mathcal{CG}$ definition documents before transmission and sending them as SOAP attachments rather than in message bodies.

8. Conclusions and Future Work. This paper describes the role of XML when communicating Condensed Graph definitions between WebCom machines. The determining factors in choosing XML for communications are described, and empirical tests illustrate the bandwidth savings observed in using XML instead of Java object serialization. XML has a number of other benefits over object serialization in that it is more extensible; extra graphs can be easily added to an XML file, additional elements and attributes can be added to XML files, without breaking backwards compatability, to support metadata such as layout information particular to the IDE and scheduling policies for nodes.

In the current version of WebCom, only graph definitions and serialized V-Graphs are stored in XML files. Future versions of the $\mathcal{CG}$ XML file format will facilitate the specification of metadata as described above. In addition, other policies can be included that temper the bahaviour of the metacomputer, allowing for different models to be loaded and used for different parts of the graph. For example, a graph could direct specific security manager, load balancing or shceduling modules to be used.

REFERENCES

[1] A. D. Birrell and B. J. Nelson, *Implementing remote procedure calls,* ACM Transactions on Computer Systems, 2(1):39–59, February 1984.

[2] Open Group Product Documentation, *DCE 1.2.2 Introduction to OSF DCE,* November 1997.

[3] The Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6,* December 2001.

[4] Microsoft Corporation, *DCOM Technical Overview*, November 1996.

[5] Sun Microsystems, *Java Remote Method Invocation—Distributed Computing for Java, White Paper.*

[6] Andrew Troelsen, *C# and the .NET Platform,* June 2001.

[7] Etham Cerami, *Web Services Essentials,* O'Reilly & Associates, March 2002.

[8] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler, *Extensible Markup Language (XML) 1.0 (Second Edition),* October 2000.

[9] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer, *Simple Object Access Protocol (SOAP) 1.1,* May 2000.

[10] Dave Winer, XML-RPC specification.

[11] Microsoft Corporation, *BizTalk Framework 2.0: Document and Message Specification,* December 2000.

[12] Dario Laverde, *Developing Web Services with Java APIs for XML (JAX Pack),* Syngress, March 2002.

[13] Larry Smarr and Charles E. Catlett, *Metacomputing,* Communications of the ACM, 35(6):44–52, 1992.

[14] I. Foster and C. Kesselman, *The Globus Project*, a status report. In Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop, pages 4–18, 1998.

[15] A. Grimshaw and W. Wulf et al, *The Legion vision of a worldwide virtual computer,* Communications of the ACM, 40(1), January 1997.

[16] John Morrison, Brian Clayton, David Power and Adarsh Patil, *WebCom-G: Grid enabled metacomputing,* The Journal of Neural, Parallel and Scientific Computation, Special Issue on Grid Computing, 2004.

[17] J. R. Gurd, C. C. Kirkham, and I. Watson, *The manchester prototype dataflow computer,* Communications of the ACM, 28(1):34–52, January 1985.

[18] Arvind and Kim P. Gostelow, *A computer capable of exchanging processors for time,* In Proceedings of IFIP Congress 77, pages 849–853, Toronto, Canada, August 1977.

[19] John P. Morrison, *Condensed graphs: Unifying availability-driven, coercion-driven and control-driven computing,* October 1996.

[20] Frank Harary, Robert Norman, and Dorwin Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs,* John Wiley and Sons, 1969.

[21] Rinus Plasmeijer and Marko van Eekelen, *Functional Programming and Parallel Graph Reduction,* Addison-Wesley, 1993.

[22] John P. Morrison, David A. Power, and James J. Kennedy, *An Evolution of the WebCom Metacomputer,* The Journal of Mathematical Modelling and Algorithms: Special issue on Computational Science and Applications, 2003(2), pp 263-276, Editor: G. A. Gravvanis.

[23] John P. Morrison, James J. Kennedy and David A. Power, *Extending WebCom: A Proposed Framework for Web Based Distributed Computing,* Workshop on Metacomputing Systems and Applications, ICPP2000.

[24] John P. Morrison, James J. Kennedy, and David A. Power, *WebCom: A Volunteer-Based Metacomputer,* The Journal of Supercomputing, Volume 18(1): 47-61, January 2001.

[25] John P. Morrison, James J. Kennedy, and David A. Power, *WebCom: A Web-Based Distributed Computation Platform,* Proceedings of Distributed computing on the Web, Rostock, Germany, June 21–23, 1999.

[26] John P. Morrison, Padraig J. O'Dowd, and Philip D. Healy, *Searching RC5 Keyspaces with Distributed Reconfigurable Hardware,* ERSA 2003, Las Vegas, June 23-26, 2003.

[27] David A. Power, *WebCom: A Metacomputer Incorporating Dynamic Client Promotion and Redirection,* M. Sc. Thesis. Submitted to the National University of Ireland, July 2000, July 2000.

[28] John P. Morrison and David A. Power, *Master Promotion & Client Redirection in the WebCom System,* Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vagas, Nevada, June 26–29, 2000.

[29] John P. Morrison, Keith Power, and Neil Cafferkey, *Cyclone: Cycle Stealing System,* Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vegas, Nevada, June 26–29, 2000.

[30] Roger Riggs, Jim Waldo, Ann Wollrath and Krishna Bharat, *Pickling state in the Java^{TM} system,* In Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies, 1996.

[31] WebMethods, Inc., *Glue 5.0.2 User Guide,* June 2004.

[32] Hartmut Liefke and Dan Suciu *XMill: an efficient compressor for XML data,* In Proceedings of the 2000 ACM SIGMOD Conference on Management of Data, Dallas, Texas, 2000.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: August 15, 2002
Accepted: December 15, 2002

Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 6, Number 1, pp. 45–56. http://www.scpe.org

IMPACT OF REALISTIC WORKLOAD IN PEER-TO-PEER SYSTEMS A CASE STUDY: FREENET

DA COSTA GEORGES* AND OLIVIER RICHARD*

Abstract. This article addresses the problem of the study of the performance evaluation and behavior of the large scale Peer-to-Peer file sharing systems. In particular the impact of realistic workload is considered by evaluating the Freenet system. This evaluation is achieved by a simulation approach. A set of inputs is determined as well as their distribution law in order to generate a more realistic workload. One of them is an original characterization of user's requests. An other contribution is to show the impact of these more realistic inputs on the overall system performances. Notably new abrupt behaviors in the learning process are described.

Key words. Freenet, Stockage, Pair á Pair, Simulation, Zipf Law

1. Introduction. Large peer-to-peer file-sharing systems had became widespread thanks to the use of projects like Napster [27] and Gnutella [16]. Their simplicity for exchanging music file explains their success as well as the copyright law transgression.

The first generation systems were based on rudimentary architecture and protocols: a central index for Napster and an inundation search protocol for Gnutella. Those systems developed fast thanks to their match of the user's demands. On the other hand, the scalability challenge they bring has raised a great interest. Numerous researches address this issue. Amongst the projects, one can enumerate academic ones like Oceanstore [4], CAN [26], Past [18], Chord [29] as well as some free or commercial softwares [24, 12].

A general problem encountered is the performance evaluation of these systems and the analyze of their statistical properties. Relatively few studies are made in this area [23, 13]. The difficulty to experiment those systems in real conditions is due to the great number of computers involved. In this context it is very difficult to evaluate a prototype. Usually studies approach the evaluation problem with restricted simulations, notably little realistic workload [8, 32].

Our main motivation is to evaluate the impact of a more realistic workload on Peer-to-Peer system's performances and behavior. As it's nearly impossible to use test such systems in a real environment (from some thousand to a few millions units) and to make an evaluation fully controlled we needed to choose an alternative solution. In this first step we have chosen to focus on the Freenet system [8] and to adopt a simulation approach.

The Freenet system is a distributed file sharing system focusing on the anonymity of it's users which aims to provide an alternative uncensored Web.

The paper is organized as follows. The next section is a fast overview of the Freenet system. Section 3 describes the simulation methodology we used. Section 4 consists in a characterization of the inputs for the simulation. The section 5 presents results obtained, and the Section 6 concludes.

2. Freenet System. The Freenet Project [7] has been created in the end of the years 90 in the academic environment and is beside the first of such systems which has been the topic of a study [8].

It's aim is to provide a service of data sharing in a cooperative way while keeping a total anonymity for the author and the reader. In the same way it is aimed to prevent any kind of censorship and to become a *free web*. Users of this system are even protected from legal attacks because the data they keep on the space they share is encrypted to prevent them from being able to read them.

Freenet has a Peer-to-Peer architecture. The users share some resources to store the files and manage the routing messages. Files are referenced by keys which only depend of their content. They are spread thanks to a cache system that react dynamically according to the load.

2.1. Routing mechanism. Each node has a dynamic table which keeps the informations it possesses on the other nodes, the key they own for example. It's by providing the key associated with an object that the system can find this object. To do so, the key is forwarded from neighbor to neighbor. Each node locally compares the key with the key it knows it's neighbor own. Then it forwards the request to the one which possesses the nearest key. The requests have a limited time to live (like the *Time To Live* of IP), the HTL for *Hop To Live* which is decreased each time the requests go through a new node. The number of node a request has go through is called the request path length.

*Laboratoire ID-IMAG (UMR 5132), Projet APACHE (CNRS/INPG/INRIA/UJF) Georges.Da-Costa@imag.fr, Olivier.Richard@imag.fr)

When an object is found, it follows backward the path the request followed to find it. Each time it passes on a node, the node adds a reference to the node which had the object and mirrors the object.

This routing algorithm has some side effects. It makes the routing becoming better and better for two reasons:

- Nodes should eventually specialize in an interval of key. If a node is associated with a key in a routing table, it will tend to treat similar key.
- Nodes should specialize in stocking objects with similar keys. As forwarding a key leads to take in the cache the object, routing similar key leads to own objects with similar keys.

Those two effects should lead to improve the quality of the system by a learning process. It behaves like a cache system. The most popular objects will be more often copied and thus will spread quickly in the network and be easily reachable.

The use of caches doesn't prevent from the disappearance of files in the network. As the size of caches are finite, when a node needs to stock a new object which would leads to exceed the size granted to the cache, the system must cleans it's cache. The policy used in Freenet (we are not based on the latest version) is to destroy the oldest used object (LRU policy). It's the same algorithm which is used for the dynamical routing table. So it's not really a cache system because there is no original version of each file. Yet all the files will remain in the network as long as they will be requested.

3. Methodology of simulation. The simulation problem is difficult in the context of Peer-to-Peer systems on account of the great number of nodes in those networks and the complexity of their behavior. Moreover we aim to study the application protocol as well as it's physical impact on the network. The main constrains encountered in this domain are the time and memory. The usual approach by using the event driven simulator like NS2 [14] is not adapted to go beyond several thousand nodes in account of the cost of a packet-level simulation. To resolve those issues we did several hypotheses in order to achieve an acceptable simulation method:

- There is no temporal consideration during the simulation: each event is treated sequentially without dating.
- The application protocol traces are sufficient to obtain inputs for a simulation of the physical network.

The first hypothesis simplifies the detail level.

The second lead to decreases the complexity of the simulator by splitting it into parts: An application-level simulator and a physical-level one.

All the facets of the system behavior can't be deeply studied. Notably all the results achieved by this approach are to be considered in a statistical point of view. By example the congestion phenomena in the physical network can't be explored without temporal considerations. But we can estimate the overall workload on the physical network.

3.1. The simulator description. The resulting simulator has three parts. A *request generator*, a *logical simulator* and a *physical simulator*. For a simulation, the first part describes the inputs of the system like the file size distribution which will be explained in section 4. The two other parts simulate the core system.

The logical simulator takes care of the application-level part. It has been obtained by expanding the simulator used by the Freenet authors [8]. Some of the modifications were instrumentation aimed to extract information needed for the physical simulator part.

The physical simulator consists in routing the application level packets into the physical network. The routing algorithm used is the classical A-Star shortest path algorithm [17]. It's complexity is exponential in time according to the size of the network.

4. Realistic inputs generation. The goal of this study is to take into account the main characteristics of real inputs and their implication on the performances and the behavior of the system. In this section we select some relevant inputs and analyze their distribution law.

As the Freenet system is to provide an anonymous alternative to the Web, the chosen inputs are those relative to the HTTP traffic. The most important components are the way the requests are done and the files they aim. If those Peer-to-Peer system spread in a wide way, we think the users will use them like they actually use the web. Moreover the underlying physical structure of the Internet is needed to evaluate their performances.

To determine those input parameters we used some studies on the protocols TCP [25], HTTP [2, 28], and on the web itself [3, 1, 11, 10, 6]. Thus we used web logs [31] and specific traces extracted from the Gnutella Peer-to-Peer system.

The following deals with the requests and files characteristics, and the network structure.

TABLE 4.1
Distribution law of requests parameters. The distribution law of the file requested *is a representation of the popularity of the files, the* user's activity *represents the distribution of the number of requests send by the users, and the* temporal distance *is the time between two requests are done (this variable is not used in this study).*

	File requested	User's activity	Temporal distance
Distribution law	Extended Zipf law $P(t) = \frac{C}{t^\alpha}$, $\alpha = 0.8$	Exponential $P(t) = \frac{1}{\beta} exp(-\beta t)$	Pareto $P(t) = \alpha k^\alpha t^{-(\alpha+1)}$

4.1. Request characterization. To define a request, three pieces of information are needed: who made it, which object is requested, and when the request is made. The table 4.1 presents the retained distribution laws.

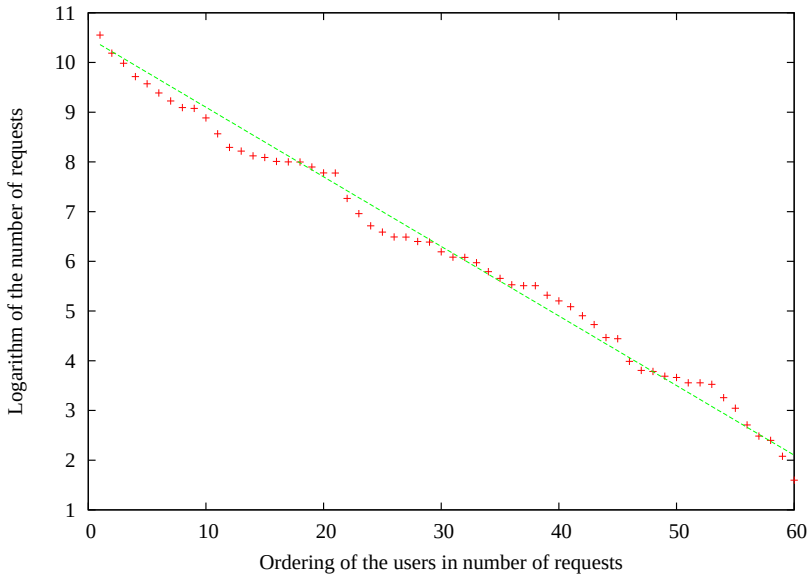


FIG. 4.1. *Distribution law of the activity of the users. The users are sorted in relation to their activity. The logarithm of the number of the request they send and a line are shown*

Requested object. According to studies of traces [3], the distribution law of the objects aimed by the requests only depends on the popularity of the files. If you suppose you can sort the files by their popularity, the request distribution follow the Zipf law [11] $P(t) = C/t$ where C is a normalization constant (used to have a total probability of one), and t is the index of popularity of the object. This law is often expressed by recalling the *90/10* law: 10% of the most requested files generate 90% of the requests. According to [6] there are no interrelationship between objects size and their popularity.

Some finer studies [6] of the web logs generated by proxies and routers show the requests repartition follows a slightly different law: $P(t) = C/t^\alpha$ where $\alpha = 0.8$. This law is called extended Zipf law.

The user who sent the request. It's quite hard to find studies based on the repartition of the user who send the requests. Most of the studies on the web load only take into account the impact of the requests on the server without taking care of who send them. There is no need to know who send a request to study the behavior of a server under stress. Some proxies logs [31] have been analyzed in order to have those information.

The distribution of the users in number of requests followed a decreasing exponential law (figure 4.1): $P(t) = \frac{e^{-\beta t}}{\beta}$ give the probability that the t^{th} user (in number of request sent) send a request at a given time with $\beta = 0.14$. This law has been obtained by analyzing Boeing's traces [5] Ircache [21] and Clarknet [9]. Those constants were obtained by making a linear regression in the log domain. The table 4.2 summarize the

characteristics of those traces, the obtained constants characterizing the distribution law, and the standard deviation.

TABLE 4.2
Summarize of the traces characteristics and the constants of the exponential law for users activities. Those traces have been made during Dates. *Each log consists in* Events number *requests. The β represents the value of the constant in the table 4.1. The* standard deviation *represent how accurate the values are.*

	Dates	Events number	β	Standard deviation
Boeing	Mar 1-5 1999	113926063	0.13	0.27
Ircache	Jan 24-30 2002	7084366	0.14	0.12
Clarknet	Aug-Sep 1995	3328632	0.18	0.4

Time between requests. Our simulator doesn't manage the time, so we didn't look at this variable. A complete study is done in [3]. In this study they found that the time between two requests follows a Pareto law: The probability that there is t seconds between two requests is $P(t) = \alpha k^\alpha t^{-(\alpha+1)}$ with $k = 1$ and $\alpha = 1.5$.

4.2. Characterization of the files: size and insertion. In this section we only take care of the object itself without taking into account it's treatment by the system. There are two free variables: It's size, and the way it has been inserted in the system (table 4.3).

TABLE 4.3
Distribution law of the parameters used to simulate the objects. Their Size. *And their* Initial position *which represents the number of files each node shares.*

	Size	Initial position
Distribution law	Lognormal $\frac{1}{t\sigma\sqrt{2\pi}}e^{-(\ln t-\mu)^2/2\sigma^2}$	Zipf law $P(t) = \frac{C}{t}$

Size of the shared files. Several studies [3, 2] show that the repartition of the files size transfered over the Internet follows a lognormal distribution. The probability that a file size is t is equal to $\frac{1}{t\sigma\sqrt{2\pi}}e^{-(\ln t-\mu)^2/2\sigma^2}$. σ and μ are experimental constants which depend of the file type. For the files over the Internet (mainly HTML pages, but some large files too), [3] gives $\mu = 9.3$ and $\sigma = 1.3$. All size are in kilo-bytes. During the measurement, the most often encountered file size was around $9ko$. Most of the files were of comparable size, and mainly of large size.

To extend this result, a Gnutella client has been modified. It gave data on the file size exchanged on the Gnutella network. In this network most of those files are multimedia ones. It showed that the file distribution of the files followed a lognormal law too. The only difference between this and the previous one concerns the values of the constants: μ and σ which become respectively 3580 and 4.9. Multimedia file are mostly around $3.6Mo$. Their size distribution scatters less that those generally found in the Internet.

Initial position of the files in the system. In our knowledge there are no studies of the initial file position in the Internet. But contrarily to the Internet, the way the files are inserted in the system may be important. In several Peer-to-Peer systems, inserting a file in the system generate some information that spread in the network.

Thus we need to know where the files where inserted in the system. To answer this question, we used web logs [31] on which Internet servers the files were acceded. It give us on which server the files were pushed.

Figure 4.2 has been produced from the informations from a proxy from Boeing. It shows the inverse of the file distribution according to the number of computers ordered by the number of files they possess. The plot of the files position is then a line, characteristic of a Zipf law.

The value of the coefficient has been calculated by doing a linear regression on the data. We don't have used all the data in order to calculate it: The trays visible on the figure 4.2 are provoked by the passage to the inverse of an integer function. to calculate this law of probability we have processed only the first value of each tray.

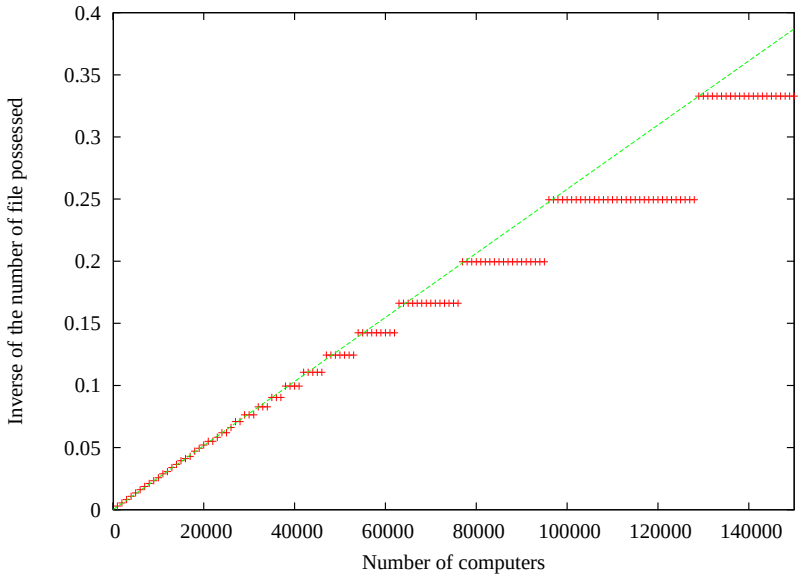


FIG. 4.2. *Inverse of the number of files possessed by the nodes. The nodes are sorted from the node that possess the most files to the node that possess the least.*

4.3. Model of physical network: Internet structure. In the context of this study, the underlying network is to be an Internet one. The modeling of the organization of the Internet network is something complex [20]. Moreover a realistic simulation should use more physical nodes than the logical ones. Everybody won't use those system. All the routers of the Internet will not participate to those systems, but they participate to the Internet structure.

Some studies have tried to characterize the Internet topology. Until now, no one found the exact nature of this structure [20]. Some characteristics have been found until now. It seems the Internet has a self-similar [10] structure. To be sharper, the Internet structure may follow some power law [22] (two functions f and g of t are link by a power law if it exists a constant α with $\forall t\ f(t) = g(t)^{\alpha}$).

We achieved a generator of network of this type by implementing the algorithm studied in [15]. It is also possible to use a more generic network generator: Brite [19]. It generate more sharp network (hierarchical one for example).

5. Results. In this section we present two kind of results. First the impact of realistic workload on the behavior of a Freenet system. It represents a deepening of the study [8]. Second an overview of the workload generated on the physical network.

5.1. Summarize of previous work. Results presented in [8] mainly show the quality of the routing algorithm and the fault tolerance of the system. They conclude that when requests are uniform, and when only the application level is taken into account, then:

- The system converges quickly (a few thousands of requests) to a stable state. At this time, more than half of the objects are under 6 hops from all the nodes.
- The system is able to grow until several hundreds thousands computers while providing a good quality of service for more than half of the objects.
- The system is fault-tolerant.
- The system evolves toward an Small World [30] architecture. In this architecture, the nodes are highly clustered yet the path length between them is small.

5.2. General experimental setup. To facilitate the comparison against [8] we use the same initial relationship network at application-level. This initial topology consists in a ring where each node is in relation with it's 4 nearest neighbors. This is a static network where all the nodes are always available.

5.3. Impact of realistic workload at the application-level. The figure 5.1 shows a direct comparison between the results of [8] which is based on the use of uniform distribution and the more realistic distributions

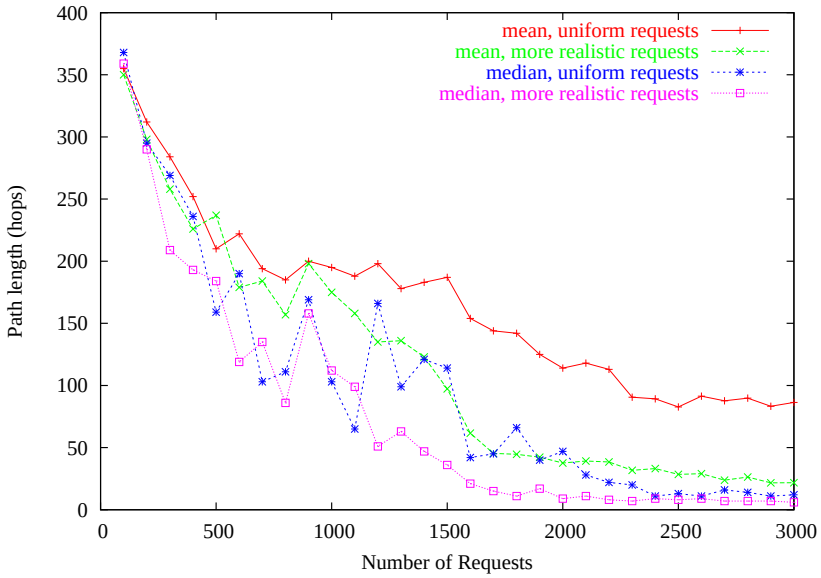


FIG. 5.1. *Evolution of the mean (or median) node number that a requests pass through to find a sought object (1000 computers) during the simulation. This evolution is evaluated with the uniform inputs, as well as with the more realistic set of inputs*

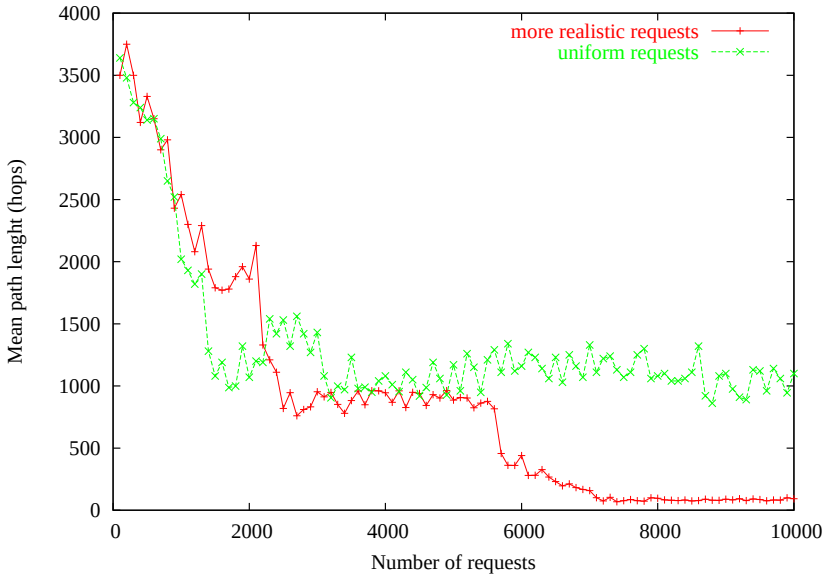


FIG. 5.2. *Evolution of the mean node number of nodes that a requests pass through to find a sought object (10000 computers) during the simulation.*

previously selected.

Those tests were done with a network of 1000 nodes, a reference cache size at 200 entries, and a file cache with 50 entries. The HTL is set at 20.

The plots on the figure 5.1 depicts the mean or median number of hops needed to access a file in function of the number of requests. Intuitively they show the evolution of the learning process. At the beginning of the learning process the system is unable to reach nearly all the files, and the more requests are treated, the more reachable the files are. There is a self organisation of the system that leads to a decrease of the distance to the objects.

At the end of the learning phase (around the 3000^{th} request) the realistic distribution leads to a significant

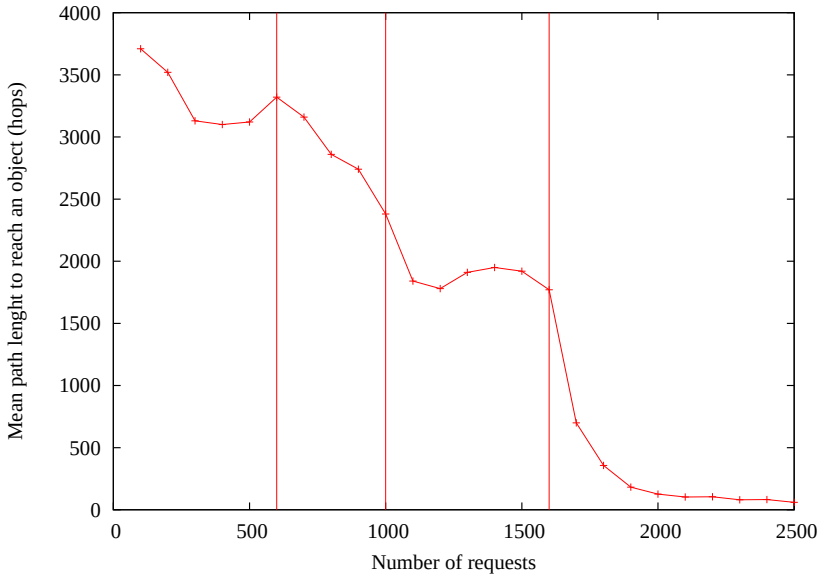


FIG. 5.3. *Evolution of the mean (or median) node number that a requests pass through to find a sought object (10000 computers) during the simulation with realistic inputs.*

improvement of the mean value of the request path length (hops) against the uniform model. For the median, there is still a slightly advantage for the realistic model (4 hops against 6).

The popularity property explains this improvement. As the most popular files are very often requested, they are wildly spread and thus the number of their access minimizes the weight of requests made on unpopular and hardly reachable files.

Contrarily to homogeneous requests that lead to a scattering of all data since the beginning. No data are penalized in this case but cost to reach data decreases little by little for all the data.

5.4. Abrupt behavior in the learning process. From this point on, we mainly focus on a larger system. The network is a 10000 nodes one. The internal value are set at 80 for the HTL, average reference cache size at 500 entries, file cache size at 2Go. The distribution used are the one of a HTTP context.

The figure 5.2 shows the mean request path length (hops) in function of the number of requests for the two approaches: the uniform and the more realistic point of view.

As on the previous experiment there is a great gap between the values at the end of the learning phase of the two approaches. But there are abrupt behaviors in the plot of the realistic requests. Those phenomena correspond to a significant decrease of the request path length. They occur after around the 2000^{th} and the 5500^{th} requests.

The behavior of the most connected nodes might give an explication of this abrupt behavior. Due to the small-world architecture of the application-level network, those nodes own most of the knowledge of the network, what corresponds to the most populated routing table.

The figures 5.3 and 5.4 underline the significant contribution of the most connected node in the learning process. The first figure is a plot of the evolution of the request path length. There are still 10000 nodes but the cache size is increased to 2500. The immediate effect of this increase is to speed up the learning phenomena which correspond of an horizontal compression of the plot.

The figure 5.4 depicts the evolution of the size of the clusters of distinctly reachable nodes by the three most connected nodes. To generate those plots, at each requests, the most connected node is selected. The number of node it can reach is plotted. Then those nodes are taken out of the network. This operation is repeated for the two next more connected nodes. In this figure the vertical lines localize the fusion of cluster of distinctly reachable nodes. Those verticals correspond in the figure 5.3 to the abrupt improvement of the request path length.

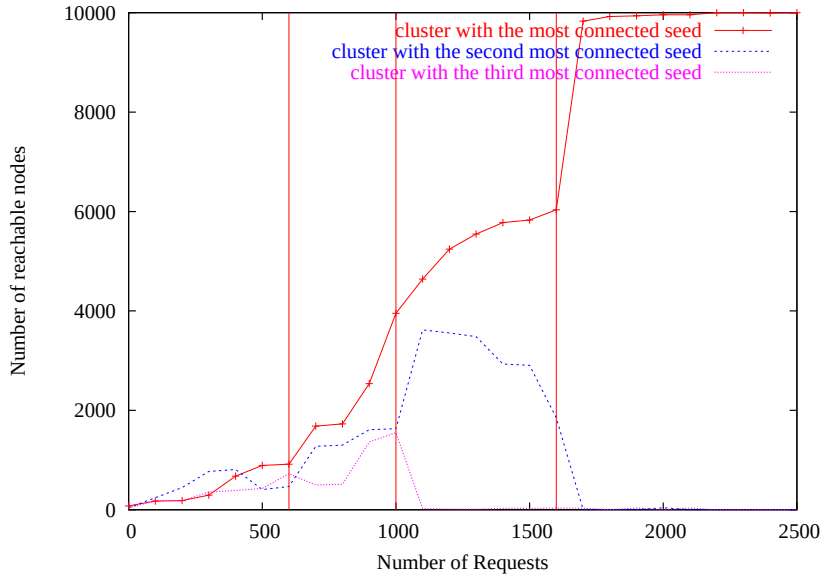


FIG. 5.4. *Clustering evolution. Evolution of the parts of the system which are distinctly reachable by the most connected nodes.*

5.5. System behavior after the learning process. The behavior of a Freenet system can be split in two parts. The part that has been approached until now: the learning process, and the stable behavior. Indeed there is a phase where the mean distance between users and files are quite constant. This is true when the file popularity is constant. Thus to verify the real quality of the learning process, it is necessary to test the system with others data than the one used for the learning phase.

To make this test several data set have been used. The firsts test are done by changing only the popularity of present files. In this category there are the full randomize test which just randomize the popularity of the file, and the reverse test which reverse the popularity of files (the most popular file become the least, and so on).

The second test is done by creating a completely new set of data.

Each test has been with 10 different runs, 100 times for each run (except for reverse which has been done only 1 time for each run).

TABLE 5.1
Impact of the popularity changes on the quality of the system

	Mean value	Standard deviation	Median value	standard deviation
Stable phase	10		4	
Reverse	10	0.008	4	0.005
Randomize	10	0.005	4	0.003
New set	10	0.013	4	0.008

The table 5.1 shows the impact of the popularity changes on the mean and median path length between the users and the data. It shows that when the learning phase is done, the network is able to handle any repartition of popularity for the files.

This property comes from a property of SHA-1 which is used to generate the keys. When it is used on a file to generate a key, the key is anywhere in the key-space. So the first keys which are the most used are randomly dispersed. Thus as the system learn by routing the keys, it can eventually handle any key with the same efficiency.

5.6. Fault tolerance of the system. The Freenet system relies on the links between nodes to find the files. This knowledge is contained in the nodes. A qualities of such system is not to be broken if some nodes

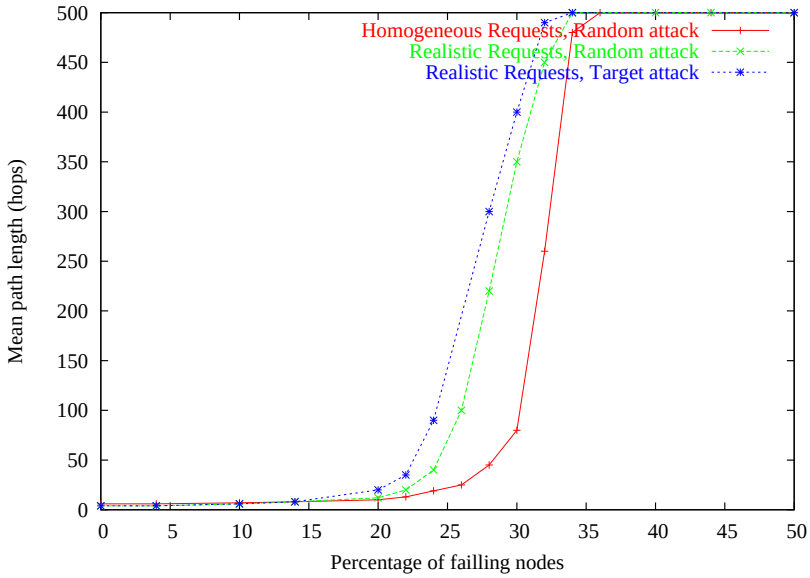


FIG. 5.5. *Comparison of the capacity of Freenet system to work when some node definitively disappear.*

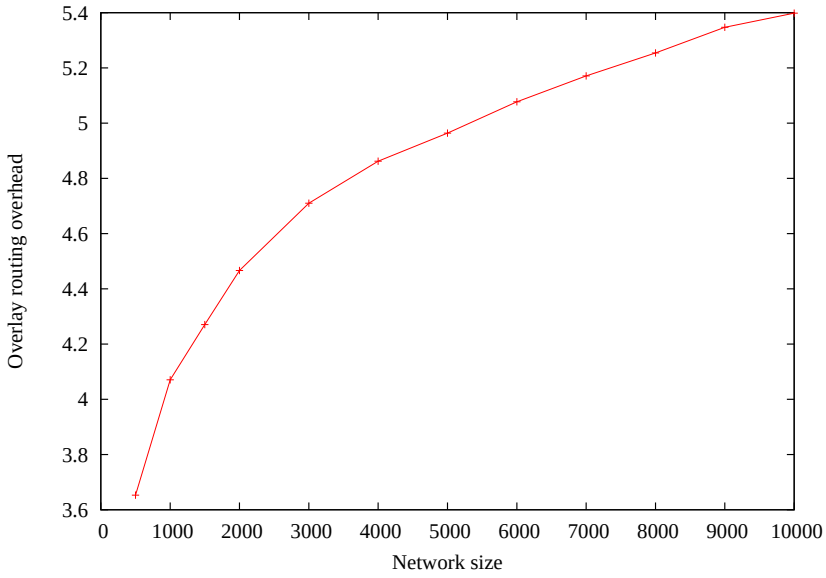


FIG. 5.6. *Evolution of the communication overhead in function of the size of the network.*

are disconnected. The case of simultaneous node failure is not just a theoretical possibility. It occurs sometime when a LAN is disconnected from the Internet, or when there is a major network partition.

There are two case: random failure and target failure. The random failure consist in choosing randomly some nodes and removing them. The target attack consist in choosing the most connected nodes (those which owns most of the knowledge of the network).

The figure 5.5 shows how the network respond for this two cases and with a network that has been created with homogeneous requests.

The system is less sensitive when it has learned with homogeneous requests. It is less sensitive too when the node are not chosen randomly. It proves that the most connected nodes are more important than others. Thus the realistic input concentrate the importance in less nodes than homogeneous input.

5.7. Workload generated on the physical network. The figure 5.6 shows the relation between the size of the network and the overlay routing overhead. It compares the length of the path followed by the files at the application level with the path they really go through at a physical level. It shows the evolution of this overhead for a network from 1000 nodes to a 1000 nodes one.

This plot show that the most node a network has, the most significant will be the overhead of the physical routing compared to the application-level view. This overhead is quite important in the Freenet case compared to some other systems (Tapestry has an overhead of less than 2 [32]).

6. Future outlook. In order to generate the requests we had to create a model of the behavior of the system's users. The behavior of users sharing multimedia files is not the same as those who share literature. There are several different populations that tend to comunicate far more in their community than with the outer world.

7. Conclusion and discussion. This paper addresses the impact of realistic workload in the Freenet Peer-to-Peer system.

Our first main contribution is to have determined a set of inputs and their distribution law in order to generate more realistic workload. This is an advance compared to the hypothesis made in previous works. Among other, we propose an original characterization of the requests generated by the users.

The second contribution is to have shown the impact of this new more realistic inputs on the overall system performance. In particular we note a great decrease of the mean request path length with more realistic inputs. An other noteworthy point is the presence of an abrupt behavior into the learning process in this context.

We have also present the cost of the overlay routing overhead in Freenet systems. This overhead is in part due to the Freenet protocol which doesn't take care of the quality of the connections it uses.

Those are preliminary results which still have to be consolidate. First of all, we have used a simulation hypothesis (time independence) to make possible the study with several thousand nodes. This hypothesis is to be validated. Two approaches might be considered: a more detailed simulation with less nodes, and an execution driven by traces. Then the physical approach is to be extended to a more realistic network.

Another important point would be to consider the intrinsic dynamic characters of Peer-to-Peer systems. We think the most important are availability of nodes and the evolution of the files popularity which will be one of our main research field in the near future.

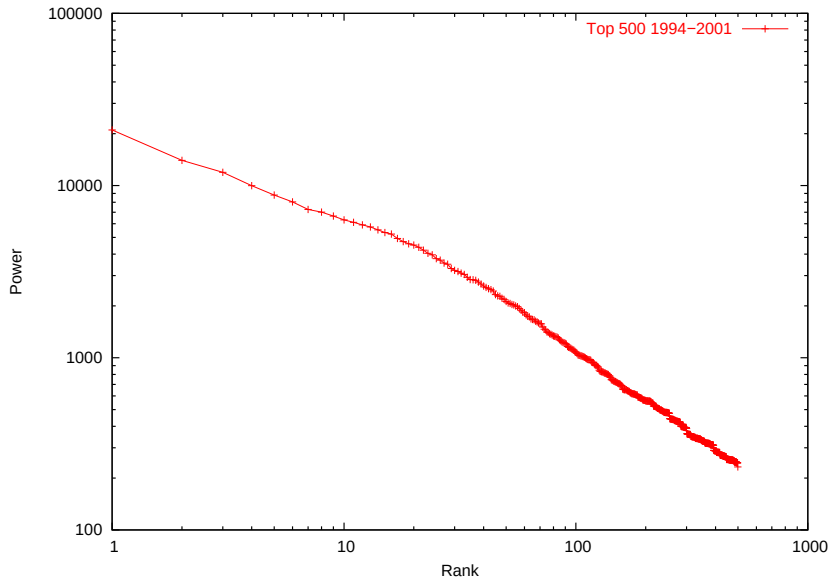


FIG. 8.1. *Logarithmic view of the rank of computers of a given power in the TOP500 between the years 1994 to 2001*

8. The omnipresent Zipf Law. In this article we have encountered several time the zipf law, or sometimes the extended zipf law. It occurred when we had to model the network, or the popularity of files. The figures 8.1 and 8.2 shows that the power of nodes are following the same law. This law is omnipresent when

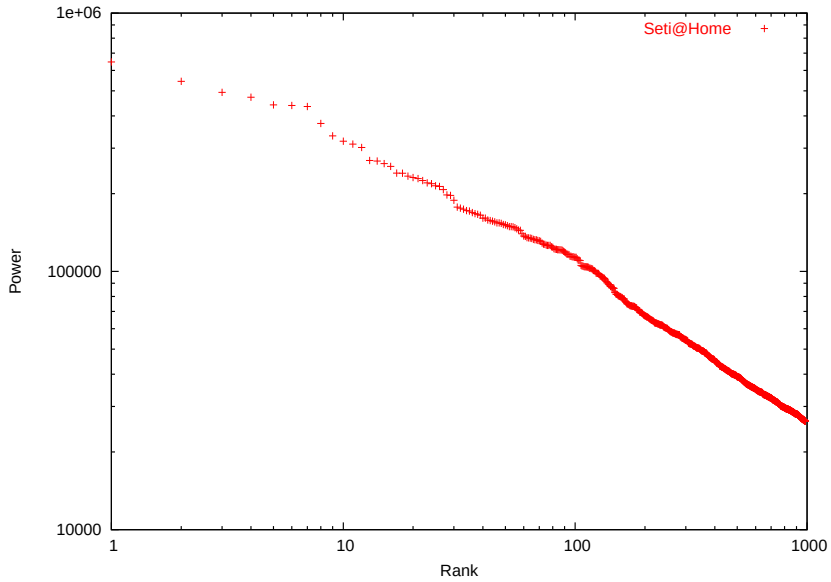


FIG. 8.2. *Logarithmic view of the most powerful participants during the 2002 summer*

there is a need to dimension hardware characteristics such as the power, network bandwidth, and so on. This law often occurs in some other fields like natural language [33].

Acknowledgments. We would like to thank the ID laboratory for granting access to its Cluster Computing Center *http://www-id.imag.fr/grappes.html*. This work utilized the ID/HP i-cluster. We would like to thank *ircache*, *Boeing* and *clarknet* for allowing us to use their web logs.

REFERENCES

[1] V. ALMEIDA, A. BESTAVROS, M. CROVELLA, AND A. DEOLIVEIRA, *Characterizing reference locality in the WWW*, Tech. Report 1996-011, Boston University Computer Science Department, 21, 1996.
[2] M. F. ARLITT AND C. L. WILLIAMSON, *Web server workload characterization: The search for invariants*, in Measurement and Modeling of Computer Systems, 1996, pp. 126–137.
[3] P. BARFORD AND M. CROVELLA, *Generating representative web workloads for network and server performance evaluation*, in Measurement and Modeling of Computer Systems, 1998, pp. 151–160.
[4] H. C. P. BEN Y. ZHAO JOHN KUBIATOWICZ, *Silverback: A global-scale archival system*, tech. report, Computer Science Division University of California Berkeley, 2001.
[5] `http://repository.cs.vt.edu/`
[6] L. BRESLAU, P. CAO, L. FAN, G. PHILLIPS, AND S. SHENKER, *Web caching and zipf-like distributions: Evidence and implications*, 1998.
[7] I. CLARKE, *A distributed decentralised information storage and retrieval system*, 1999.
[8] I. CLARKE, O. SANDBERG, B. WILEY, AND T. W. HONG, *Freenet: A distributed anonymous information storage and retrieval system*, in Workshop on Design Issues in Anonymity and Unobservability, 2000, pp. 46–66.
[9] `http://ita.ee.lbl.gov/html/contrib/clarknet-http.html`.
[10] M. E. CROVELLA AND A. BESTAVROS, *Self-similarity in World Wide Web traffic: evidence and possible causes*, IEEE/ACM Transactions on Networking, 5 (1997), pp. 835–846.
[11] C. R. CUNHA, A. BESTAVROS, AND M. E. CROVELLA, *Characteristics of www client-based traces*, Tech. Report TR-95-010, Boston University Computer Science Department, 1995.
[12] `http://www.fasttrack.nu/`.
[13] I. FOSTER, *Peer-to-peer architecture case study: Gnutella network*, tech. report, Computer Science Dep of the University of Chicago, 2001. TR-2001-26.
[14] FOURTH IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (ISCC'99), *Network Emulation in the Vint/NS Simulator*, July 1999.
[15] GLOBECOM '2000, *Generating network topologies that obey power laws*, November 2000.
[16] `http://gnutella.wego.com/`.
[17] P. HART, N. NILSSON, AND B. RAPHAEL, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics, 2 (1968), pp. 100–107.
[18] G. HEIDELBERG, ed., *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), November 2001.

[19] IEEE MASCOTS'01, *BRITE: An Approach to Universal Topology Generation*, 2001.
[20] IEEE/ACM Transactions on Networking, *Difficulties in Simulating the Internet*, vol. 9, August 2001.
[21] `ftp://ftp.ircache.net/traces/`.
[22] A. Medina, I. Matta, and J. Byers, *On the origin of power laws in internet topologies*, Tech. Report 2000-004, Boston University Computer Science Department, January 2000.
[23] Multimedia Computing and Networking 2002 (MMCN'02), *A Measurement Study of Peer-to-Peer File Sharing Systems*, 2002.
[24] `http://www.opencola.com/`.
[25] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe, *Modeling TCP throughput: A simple model and its empirical validation*, in ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication, Vancouver, CA, 1998, pp. 303–314.
[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content addressable network*, Tech. Report TR-00-010, Berkeley, Berkeley, CA, 2000.
[27] D. Scholl, *Nap protocol specification*, tech. report, Sourceforge, 2000. `http://opennap.sourceforge.net/napster.txt`
[28] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz, *Analysis of a very large altavista query log*, tech. report, DEC, 1998. On-line at `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1998-014.html`
[29] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, Tech. Report TR-819, MIT, March 2001.
[30] T. Walsh, *Search in a small world*, in IJCAI, 1999, pp. 1172–1177.
[31] `http://www.web-caching.com/traces-logs.html`
[32] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*, tech. report, Computer Science Division University of California Berkeley, 2001.
[33] G. Zipf, *Human behavior and the principle of least effort*, 1949.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: October 02, 2002
Accepted: December 02, 2002

Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 6, Number 1, pp. 57–69. http://www.scpe.org

PARALLEL EXTENSION OF A DYNAMIC PERFORMANCE FORECASTING TOOL

EDDY CARON[†‡], FRÉDÉRIC DESPREZ[†] AND FRÉDÉRIC SUTER[†]

Abstract. This paper presents an extension of a performance evaluation library called Fast to handle parallel routines. Fast is a dynamic performance forecasting tool in a grid environment. We propose to combine estimations given by Fast about sequential computation routines and network availability to parallel routine models coming from code analysis.

Key words. Performance forecasting and modeling, parallel routines.

1. Introduction. Thanks to metacomputing [10] it is now possible to perform computations on the aggregation of machines all over the world. This is the promise of a huge computational power because many interactive machines are under-used. Moreover, computations could be performed on the most appropriate computer. But such solutions are difficult to achieve while keeping good efficiency. One of the reasons is that these machines can either be workstations, clusters or even shared memory parallel computers. This heterogeneity in terms of computing power and network connectivity implies that the execution time of a routine may vary and strongly depends on the execution platform. A trend of metacomputing, called *Application Service Providing*, allows multidisciplinary applications to access computational servers using a system based on agents. These agents are in charge of the choice of the most appropriate server and task scheduling and mapping. In such a context it becomes very important to have tools able to collect knowledge about servers at runtime to forecast the execution times of the tasks to schedule. Furthermore, in a metacomputing context, we might deal with hierarchical networks of heterogeneous computers with hierarchical memories. Even if software packages exist to acquire information needed about both computer and network using monitoring techniques [14], they often use a flat and homogeneous view of a metacomputing system. To be closer to reality we have to model computation routines with regard to the computer which will execute them.

The computational servers we target can be sequential or parallel. Moreover, they are running libraries such as the BLAS, LAPACK and/or their parallel counterparts (PBLAS, ScaLAPACK). The former libraries are using virtual grids of processors and block-sizes to distribute the matrices involved in the computations. The performance evalutation has thus to take into account the shape of the grid as well as the distributions of the input and output data. In this paper, we propose a careful evaluation of several linear algebra routines combined with a run-time evaluation of the parameters of the target machines. This allows us to optimize the mapping of data-parallel tasks in a client-server environment.

In the next section we give an overview of Fast (Fast Agent's System Timer) [5, 12], the dynamic performance forecasting tool we have extended. After a description of some related work, in section 4 we propose a combination between dynamic data acquisition and code analysis which will be the core of the extension of Fast to handle parallel routines. We detail the extension on two examples of parallel routines. For each model we provide some experimental results validating our approach. Finally in section 5 we show how can our tool be used in a scheduling context.

2. Overview of Fast. Fast is a dynamic performance forecasting tool in a metacomputing environment designed to handle the important issues presented in the previous section. Informations acquired by Fast concern sequential computation routines and their execution platforms (memory, load) as well as the communication costs between the different elements of the software architecture. Fast should be useful to a scheduler to optimize task mapping on a heterogeneous and distributed environment.

As shown in Figure 2.1, Fast is composed of several modules. The first one, launched when Fast is installed on a computational server, is a tool that executes extensive benchmarks of sequential routines on that server and then extracts models from the results by polynomial regression. The models obtained are stored in a LDAP [11] tree. The second part is the library itself called by the client application. This library is based on two modules and a user API: the *routine needs modeling* module and the *system availabilities* module. The former extracts from the LDAP tree the time and space needs of a sequential computational routine on a given machine for a given set of parameters, while the latter forecasts the behavior of dynamically changing resources, *e.g.,* workload, bandwidth, memory use, ... To acquire such dynamic data, Fast is essentially based on NWS (Network Weather

[†]ReMaP – LIP, École Normale Supérieure de Lyon. 46, Allée d'Italie. F-69364 Lyon Cedex 07.
[‡](Eddy.Caron@ens-lyon.fr) Questions, comments, or corrections to this document may be directed to that email address.

Service) [14] from the Universiy of Santa Barbara. NWS is a distributed system that periodically monitors and dynamically forecasts performance of various network and computational resources using sensors. NWS can monitor several resources including communications links, CPU, disk space and memory. Moreover, NWS is not only able to obtain measurements, it can also forecast the evolution of the monitored system.

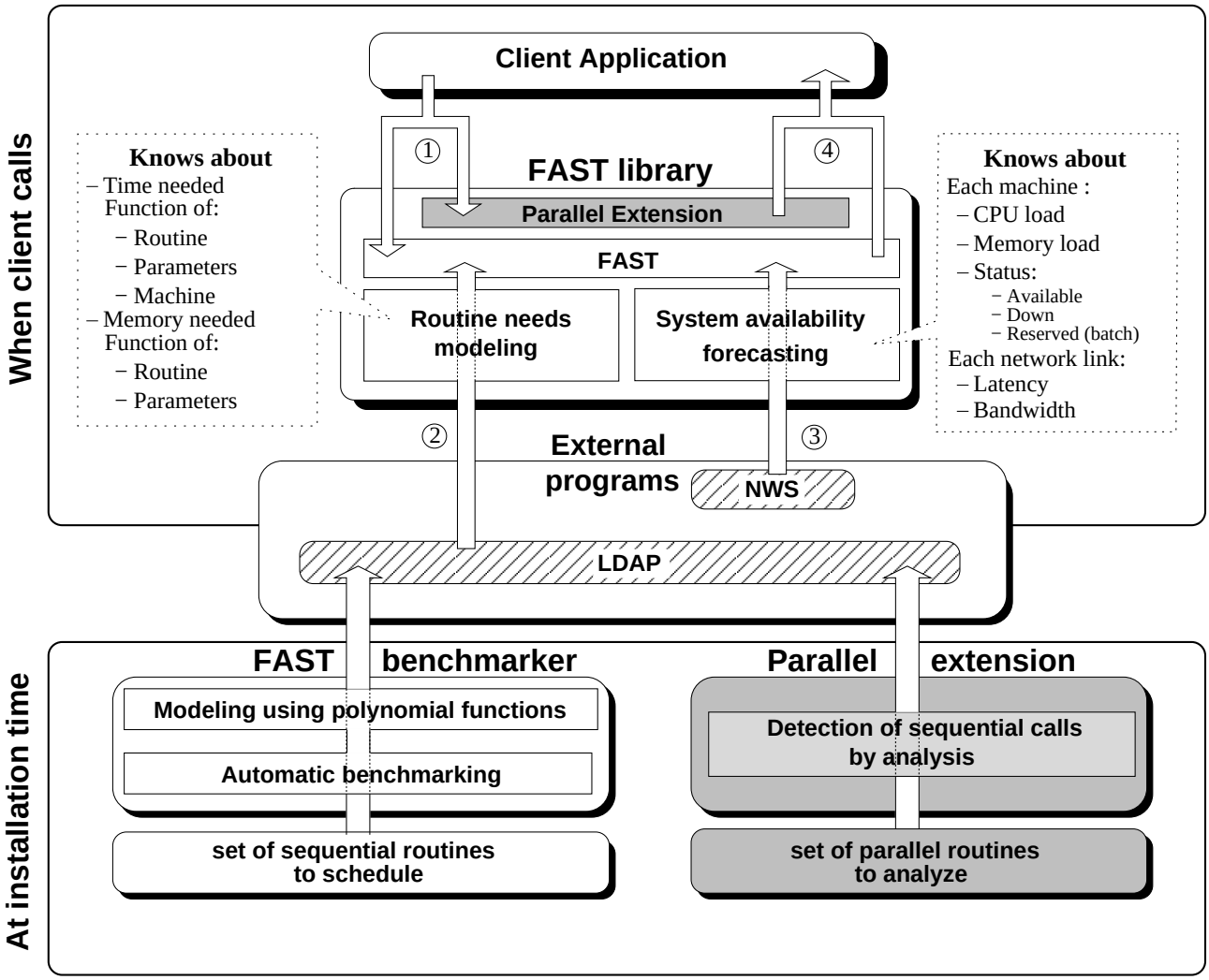


FIG. 2.1. *Overview of the* FAST *architecture with the extension to handle parallel routines.*

FAST extends NWS as it allows to determine theoretical needs of computation routines in terms of execution time and memory. The current version of FAST only handles regular sequential routines like those of the dense linear algebra library BLAS [7]. However, BLAS kernels represent the heart of many scientific applications, especially in numerical simulation. The approach chosen by FAST to forecast execution times of such routines is an extensive benchmark followed by a polynomial regression. Indeed this kind of routines is often strongly optimized with regard to the platform, either by vendors or by automatic code generation [8]. A code analysis to find an accurate model thus becomes tedious. Furthermore, those optimizations are often based on a better use of hierarchical memories. Then a generic benchmark of a computer will lack of accuracy because the performance that can be achieved on a computer is not constant as it depends on how a routine uses cache memories.

The current version of FAST does not take into account the availability of parallel versions of the routines. In this paper we focus on the integration of parallel routines handling into FAST. These routines are difficult to benchmark but easier to analyze. Indeed, they often can be reduced to a succession of computation and communication phases. Computation phases are composed of calls to one or several sequential routines while communication phases are point–to–point or global communication patterns. Timing becomes even more tedious

if we add the handling of data redistribution between servers and the choice of the "optimal" virtual processor grid where computations are performed. So it seems possible and interesting to combine code analysis and information given by FAST about sequential execution times and network availability.

The shaded parts in Figure 2.1 show where our work takes place in the FAST architecture. Our parallel extension of FAST is decomposed in two parts. The former is based on code analysis made at the installation of FAST on a machine while the latter is an extension of the API. Indeed FAST's estimations are injected into a model coming from code analysis. Once this combination performed the execution time of the modeled parallel routine can be estimated by FAST and is thus available through the FAST standard API. Both parts will be detailed in Section 4.

3. Related Work. To obtain a good schedule for a parallel application it is mandatory to initially determine the computation time of each of its tasks and communication costs introduced by parallelism. The most common method to determine these times is to describe the application and model the parallel computer that executes the routine.

The modeling of a parallel computer and more precisely of communication costs can be considered from different points of view. A straightforward model can be defined ignoring communication costs. A parallel routine can thus be modeled using Amdahl's law. This kind of model is not realistic in the case of distributed memory architectures. Indeed on such platforms communications represent an important part of the total execution time of an application and can not be neglected. Furthermore, this model does not allow to handle the impact of processor grid shape on routine performance. In such conditions, it becomes important to study the communication scheme of the considered routine to determine the best size and shape to use. For instance the matrix–matrix multiplication routine of the ScaLAPACK library, compact grids achieve better performance than elongated ones. We can see on Figure 3.1 that a too simple use of Amdahl's law, consisting in adding a processor to the execution platform while the obtained computation time is lower than the one previously estimated has a limited validity and may not detect the optimal execution platform.

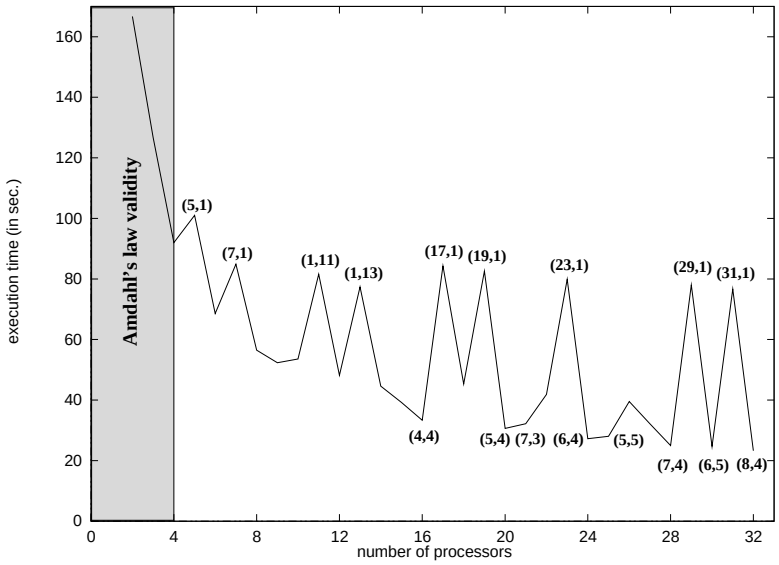


FIG. 3.1. *Execution times of a* 4096×4096 *matrix–matrix multiplication achieved on the best grid using a given number of processors.*

The *delay* [13] and *LogP* [3] models have been designed to take communications into account. The former as a constant delay d while the latter considers four theoretical parameters: transmission time from a processor to an other (L), computation overhead of a communication (o), network bandwidth (g) and number of processors (P). However, the delay model may be not accurate enough while *LogP* is too complicated to model a metacomputing platform in a simple but realistic way.

About the modeling of parallel algorithms such as those used in libraries like ScaLAPACK, several approaches are possible. In [9], authors aim at using parallel routines on clusters when it is possible to achieve better performance than with a serial execution. Their model consists in identifying the sequential calls in the

code of the parallel routine and to replace them by functions depending on data sizes and relative performance of the sequential counterpart of this routine on a node of the cluster. However, this model does not take into account the load variation of the execution platform neither of the optimizations based on pipelining made in the routines. In [6] a model of parallel routines is proposed based on estimation of sequential counterparts. The result of this estimation is a polynomial whose variables depend on the matrix sizes. Coefficients are specific to the couple {algorithm, machine}. These parameters are determined by interpolating, dimension per dimension, a set of curves obtained from the execution of the routine with small data. The *ChronosMix* environment [1] uses micro-benchmarking to estimate execution times of parallel applications. This technique consists in performing extensive tests on a set of C/MPI instructions. Some source codes, written in this language and employing that communication library, are then parsed to determine their execution times. The use of this environment is therefore limited to such codes. But most of numerical libraries are still written in Fortran. Moreover, in the particular case of ScaLAPACK, communications are handled through the BLACS library implemented on top of MPI but also on top of PVM.

4. Extension of Fast to Handle Parallel Routines.

4.1. Modeling Choices. The first version of the extension only handles some routines of the parallel dense linear algebra library ScaLAPACK. For such routines the description step consists only in determining which sequential counterparts are called, their calling parameters (*i.e.,* data sizes, multiplying factors, transposition, ...), the communication schemes and the amount of data exchanged. Once this analysis completed, the computation part can easily be forecasted. Indeed since FAST is able to estimate each sequential counterpart, FAST calls are sufficient enough to determine their execution times.

Furthermore, processors executing a ScaLAPACK code have to be homogeneous to achieve optimal performance, and the network between these processors has also to be homogeneous. It allows us two major simplifications. First processors being homogeneous, the benchmarking phase of FAST can be executed on only one processor. Then concerning communications we only have to monitor a few representative links to obtain a good overview of the global behavior.

For the estimation of point-to-point communications we chose the $\lambda + L\tau$ model where λ is the latency of the network, L the size of the message and τ the time to transfer an element, *i.e.,* the inverse of the network bandwidth. L can be determined during the analysis while λ and τ can be estimated with FAST calls. In a broadcast operation the λ and τ constants are replaced by functions depending on the processor grid topology [2]. If we consider a grid with p rows and q columns λ_p^q will be the latency for a column of p processors to broadcast their data (assuming a uniform distribution) to processors that are on the same row and $1/\tau_p^q$ will be the bandwidth. In the same way λ_q^p and τ_q^p denote the time for a line of q processors to broadcast their data to processors that are on the same column. These functions directly depend on the implementation of the broadcast. For example, on a cluster of workstations connected through a switch, the broadcast is actually executed following a tree. In this case λ_p^q will be equal to $\lceil \log_2\ q \rceil \times \lambda$ and τ_p^q equal to $(\lceil \log_2\ q \rceil / p) \times \tau$ where λ is the latency for one node and $1/\tau$ as the average bandwidth.

In the next section, we give a view of the content of the extension by detailing both examples of parallel dense matrix-matrix multiplication and triangular solve routines.

4.2. Matrix–Matrix Multiplication Model. The routine `pdgemm` of ScaLAPACK performs the product

$$C = \alpha op(A) \times op(B) + \beta C$$

where $op(A)$ (resp. $op(B)$) can be A or A^t (resp. B or B^t). In this paper we have focused on the $C = AB$ case[1]. A is a $M \times K$ matrix, B a $K \times N$ matrix, and the result C a $M \times N$ matrix. As these matrices are distributed on a $p \times q$ processor grid in a block-cyclic way with a block size of R, computation time is expressed as

$$\left\lceil \frac{K}{R} \right\rceil * \texttt{dgemm\_time}, \tag{4.1}$$

where `dgemm_time` is given by the following FAST call

```
fast_comp_time (host, dgemm_desc, &dgemm_time),
```

where `host` is one of the processors involved in the computation of `pdgemm` and `dgemm_desc` is a structure

[1] The other cases are similar.

containing informations such as the size of the matrices involved, if they are transposed or not, ... Matrices given in arguments to this call to FAST are of size $\lceil M/p \rceil \times R$ for the first operand and $R \times \lceil N/q \rceil$ for the second one.

To accurately estimate the communication time it is important to consider the `pdgemm` communication scheme. At each step one pivot block column and one pivot block row are broadcasted to all processors, and independent products take place. Amounts of data communicated are then $M \times K$ for the broadcast of rows and $K \times N$ for the broadcast of the columns. Each broadcast is performed block by block. The communication cost of the `pdgemm` routine is thus

$$(M \times K)\tau_p^q + (K \times N)\tau_q^p + \left(\lambda_p^q + \lambda_q^p\right) \left\lceil \frac{K}{R} \right\rceil . \tag{4.2}$$

This leads us the following estimation for the routine `pdgemm`

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{dgemm\_time} + (M \times K)\tau_p^q + (K \times N)\tau_q^p + \left(\lambda_p^q + \lambda_q^p\right) \left\lceil \frac{K}{R} \right\rceil . \tag{4.3}$$

If we assume that broadcast operations are performed following a tree, τ_p^q, τ_q^p, λ_p^q and λ_q^p can be replaced by their values depending on τ and λ. These two variables are estimated by the following FAST calls

```
fast_avail (bandwidth, source, dest, &tau)
```

and

```
fast_avail (latency, source, dest, &lambda),
```

where the link between `source` and `dest` is one of those monitored by FAST. Equation 4.2 thus becomes

$$\frac{\left(\frac{\lceil \log_2\ q \rceil \times M \times K}{p} + \frac{\lceil \log_2\ p \rceil \times K \times N}{q}\right)}{\texttt{tau}} + \left\lceil \frac{K}{R} \right\rceil (\lceil \log_2\ q \rceil + \lceil \log_2\ p \rceil) \times \texttt{lambda}. \tag{4.4}$$

4.3. Accuracy of the Matrix–Matrix Multiplication Model. To validate our modeling technique we ran several tests on *i-cluster* which is a cluster of HP e–vectra nodes (Pentium III 733 MHz with 256 MB of memory per node) connected through a Fast Ethernet network via HP Procurve 4000 switches. So the broadcast is actually executed following a tree.

In a first experiment we tried to validate the accuracy of the extension for a given processor grid. Figure 4.1 shows the error rate of the forecast with regard to the actual execution time for matrix multiplications executed on a 8×4 processor grid. Matrices are of sizes 1024 up to 10240. We can see that our extension provides very accurate forecasts as the average error rate is less than 3%.

Figure 4.2 presents a comparison between the estimated time given by our model (top) and the actual execution time (bottom) for the `pdgemm` routine on all possible grids from 1 up to 32 processors of *i-cluster*. Matrices are of size 2048 and the block size is fixed to 64. The x-axis represents the number of rows of the processor grid, the y-axis the number of columns and the z-axis the execution time in seconds. We can see that the estimation given by our extension is very close to the experimental execution times. The maximal error is less than 15% while the average error is less than 4%. Furthermore, these figures confirm the impact of topology on performance. Indeed compact grids achieve better performance than elongated ones because of the symmetric communication pattern of the routine. The different stages for row and column topologies can be explained by the log term introduced by the broadcast tree. These results shows that our evaluation can be efficiently used to choose a grid shape for a parallel routine call. Combined with a run-time evaluation of parameters like communication, machine load or memory availability, we can then build an efficient scheduler for ASP environments.

4.4. Triangular Solve Model. The routine `pdtrsm` of the ScaLAPACK library can be used to solve the following systems: $op(A) * X = \alpha B$ and $X * op(A) = \alpha B$ where $op(A)$ is equal to A or A^t. A is a $N \times N$ upper or lower triangular matrix which can be unitary or not. X and B are two $M \times N$ matrices.

Figure 4.3 presents graphical equivalents of `pdtrsm` calls depending on the values of the three parameters `UPLO`, `SIDE` and `TRANSA` defining respectively whether A is upper or lower triangular, X is on the left or right

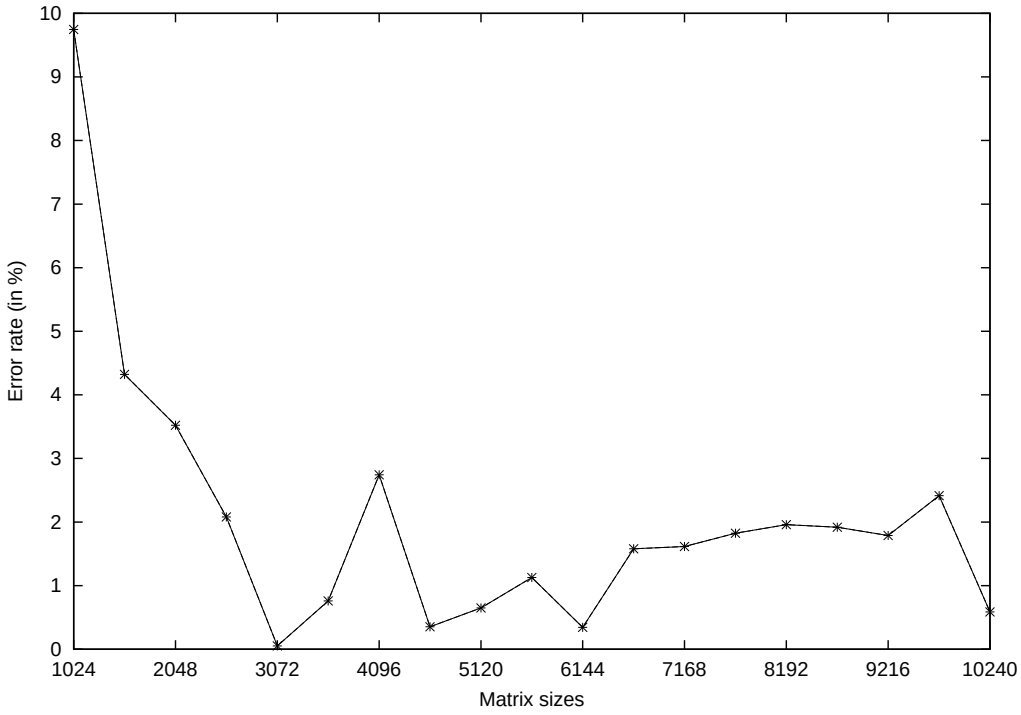


FIG. 4.1. *Error rate between forecasted and actual execution time for a matrix–matrix multiplication on a* 8×4 *processor grid.*

side of A and if A has to be transposed or not. The eight possible calls can be grouped into four cases as shown in Figure 4.3. These cases can also be grouped into two family which achieve different performance depending on the shape of the processor grid. The former includes cases 1 and 3 and will execute faster on row–dominant grids, while the latter contains cases 2 and 4 and will achieve better performance on column–dominant grids. Finally, the routine `pdtrsm` is actually composed of calls to the subroutine `pbdtrsm` which computes the same operation but when the number of rows of B is less or equal to the size of a distribution block, R. To obtain the computation cost of the `pdtrsm` routine of ScaLAPACK, the cost of a call to `pbdtrsm` has then to be multiplied by $\lceil M/R \rceil$ as each call to that routine solves a block of rows of X.

To analyze why some shapes are clearly better than some other, we focus on the case $X * A = B$ where A is a non-unitary upper triangular matrix. Figure 4.4 shows the results we obtained on 8 processors for several grid shapes. The library used is ScaLAPACK v1.6 which is based on PBLAS v1.0. In this particular case (number 2 in Figure 4.3), we can see that a 8 processors row grid (*i.e.,* 1×8) achieves performance 3.7 times better than a 8 processors column grid (*i.e.,* 8×1). We first focused our analysis on this case where the execution platform is a row, and then extended it too the more general case of a rectangular grid.

4.4.1. On a Row. To solve the system of equations presented in Figure 4.5 (where A is an $n \times n$ block matrix), the principle is the following. The processor that owns the current diagonal block A_{ii} performs a sequential triangular solve to compute b_{1i}. The resulting block is broadcasted to the other processors. The receivers can update the unsolved blocks they own, *i.e.,* compute $b_{1j} - b_{1i}a_{ij}$, for $i < j \leq n$. This sequence is thus repeated for each diagonal block, as shown in Figure 4.6.

We denote the time to compute a sequential triangular solve as `trsm_time`. This time is estimated by

```
fast_comp_time (host, trsm_desc, &trsm_time),
```

where `host` is one of the processors involved in the computation of `pdtrsm` and `trsm_desc` is a structure containing information on matrices and calling parameters. Matrices passed in arguments to that FAST call are of size $R \times R$.

Solved blocks are broadcasted along the ring but the critical path of `pdtrsm` follows also this ring. So we have only to consider the communication between the processor that computes the sequential solve and its right neighbor. As the amount of data broadcasted is R^2, this operation is then estimated by

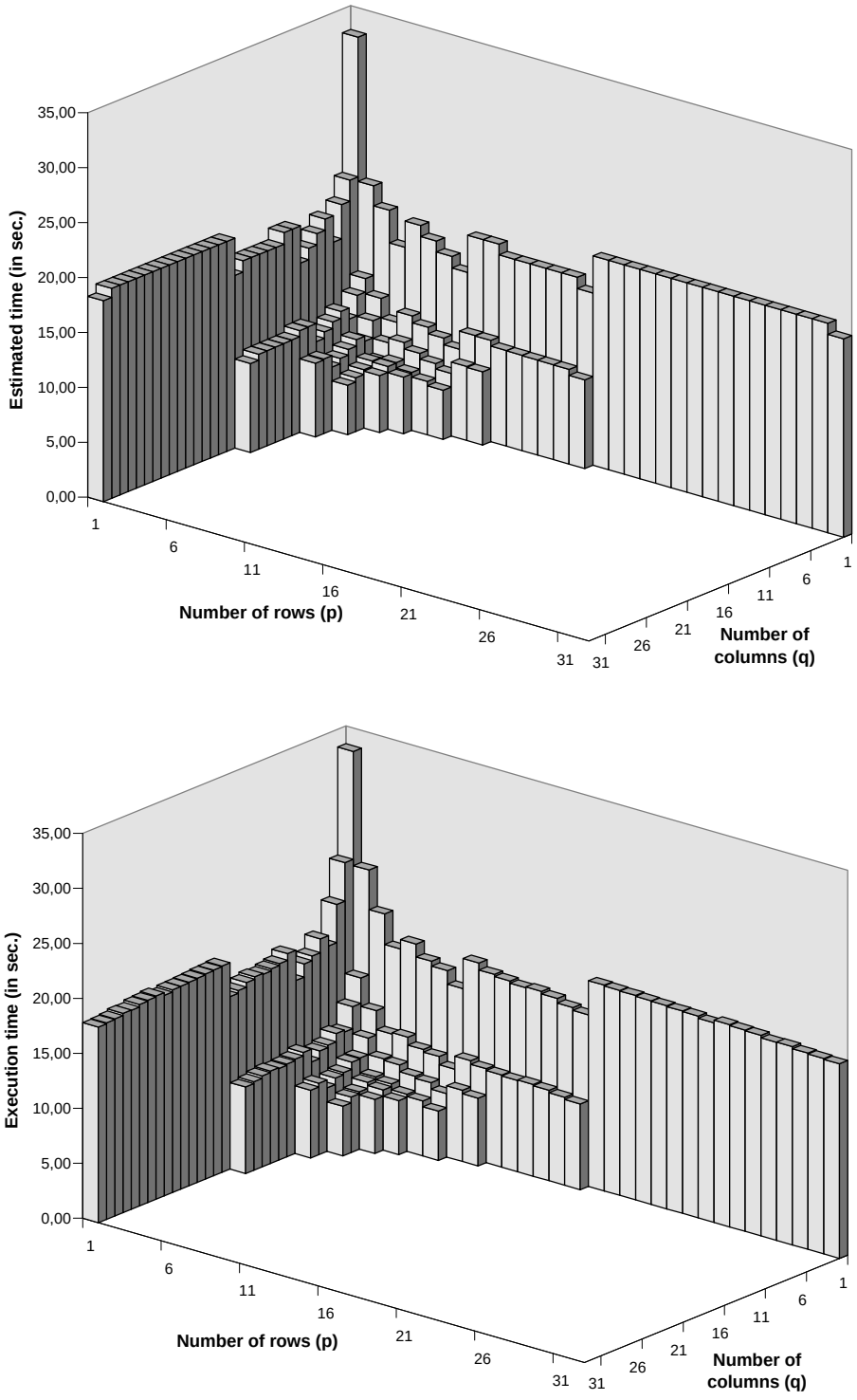


FIG. 4.2. *Comparison between estimated time (top) and actual execution time (bottom) for the* `pdgemm` *routine on all possible grids from 1 up to 32 processors of* i-cluster.

$$T_{broadcast} = R^2\texttt{tau} + \texttt{lambda}, \tag{4.5}$$

Upper / Lower	Left / Right	Transposition	Operation	Case
U	L	N	B = A \ α B	1
U	R	T		
U	L	T	B = α B / A	2
U	R	N		
L	L	N	B = A \ α B	3
L	R	T		
L	L	T	B = α B / A	4
L	R	N		

FIG. 4.3. *Correspondence between values of calling parameters and actually performed computations for the* `pdtrsm` *routine.*

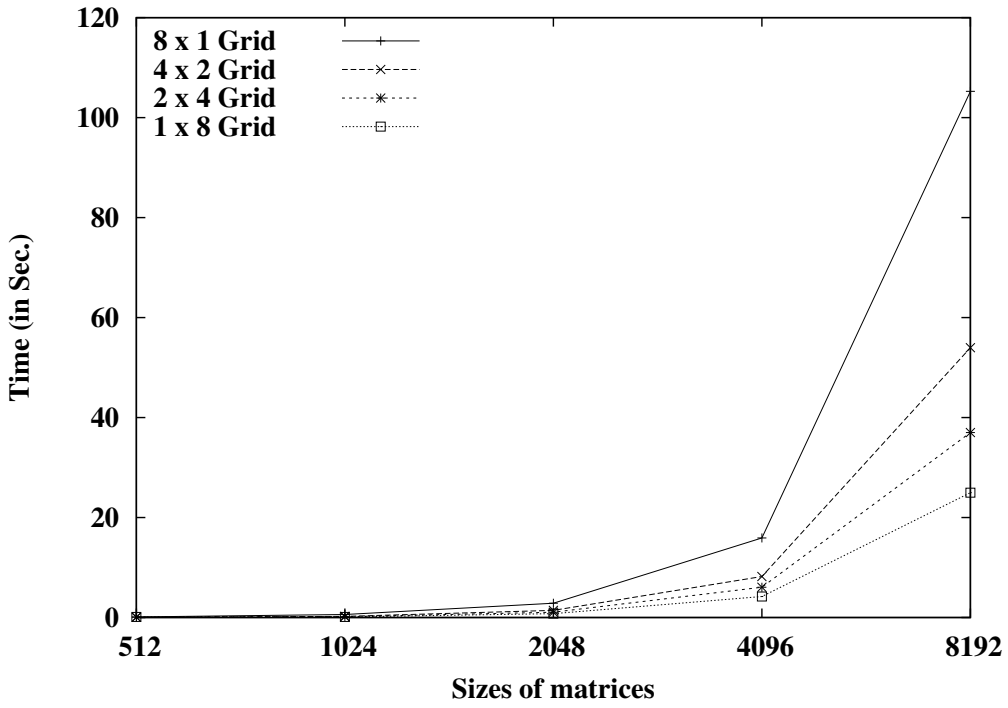


FIG. 4.4. *Performance of a* `pdtrsm` *for different grid shapes.*

where `lambda` and `tau` are estimated by the FAST calls presented in the matrix multiplication model.

At each step the update phase is performed calling the `dgemm` routine. The first operand for each of these calls is a copy of the block solved at this step and therefore is always a $R \times R$ matrix. The number of columns of the second operand depends on how many blocks have already been solved and can be expressed as $R\lceil(N - iR)/(qR)\rceil$ where i the number of solved blocks. The corresponding FAST call is then

```
fast_comp_time (host, dgemm_desc_row, &dgemm_time_row).
```

Idle times may appear if one of the receivers of a broadcast is still uptading blocks corresponding to the previous step. As our model forecasts the execution time following the critical path of the routine, to handle these idle times we apply a correction C_b as both sending and receiving processors have to wait until the

$$
\begin{aligned}
b_{11} &= b_{11}/a_{11} \\
b_{12} &= (b_{12} - b_{11}a_{12})/a_{22} \\
&\vdots \quad \vdots \\
b_{1j} &= (b_{1j} - b_{11}a_{1j} - \cdots - b_{1(j-1)}a_{(j-1)j})/a_{jj} \\
&\vdots \quad \vdots \\
b_{1j} &= (b_{1n} - b_{11}a_{1n} - \cdots - \cdots - b_{1(n-1)}a_{(n-1)n})/a_{nn}
\end{aligned}
$$

FIG. 4.5. *Computations performed in a* `pbdtrsm` *call, where A is an $n \times n$ block matrix.*

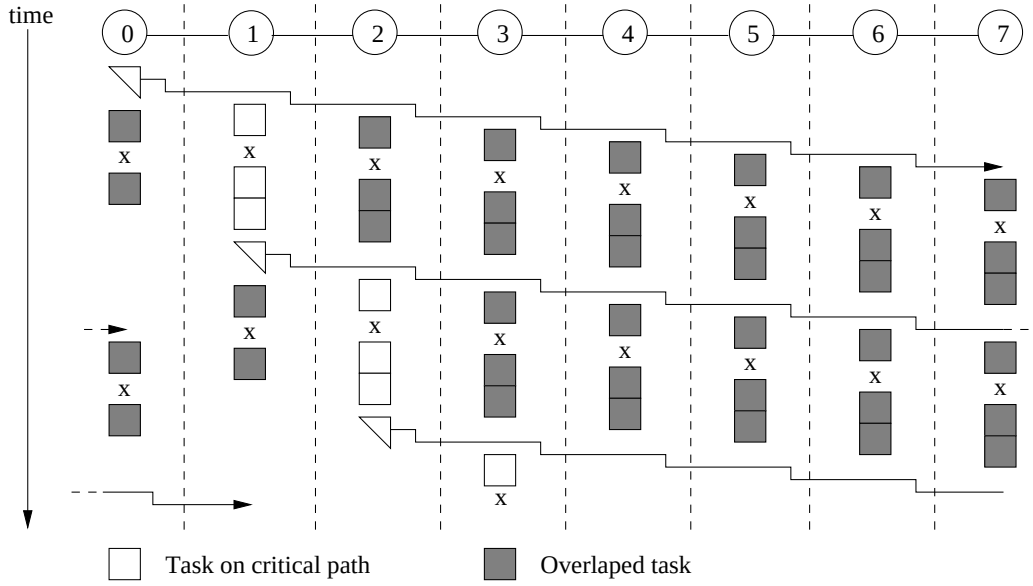


FIG. 4.6. *Execution on a ring of 8 processors of the* `pbdtrsm` *routine.*

maximum of their previously estimated times before performing this communication. Thus equation 4.6 gives the computation cost model for the `pbdtrsm` routine on a row of processors.

$$
\sum_{i=1}^{\lceil N/R \rceil} (\texttt{trsm\_time} + T_{broadcast} + C_b + \texttt{dgemm\_time\_row}) \,. \tag{4.6}
$$

4.4.2. On a Rectangular Grid. The main difference between the previous case and the general one appears in the update phase. Indeed another pipeline is introduced in the general case. It consists in splitting the update phase in two steps. The first updates the first $(p-1)$ blocks (where p is the number of processor rows of the grid) while the second deals with the remaining blocks. Once the first part has been updated, it is sent to the processor which is on the same column and on the next row. The receiving processor then performs an accumulation to complete the update of its blocks, as shown in Figure 4.7.

This optimization implies we have two values for the number of columns of the second operand of the `dgemm` calls. The former can be expressed as the minimum between $R\lceil (N - iR)/(qR) \rceil$ and $R(p-1)$ while the latter will be $R(\lceil (N - iR)/(qR) \rceil - (p-1)))$ if positive. The first operand is still a $R \times R$ matrix. We then have two FAST calls to estimate these two different matrix products of the update phase.

```
fast_comp_time (host, dgemm_desc_1, &dgemm_time_1),
```

and

```
fast_comp_time (host, dgemm_desc_2, &dgemm_time_2).
```

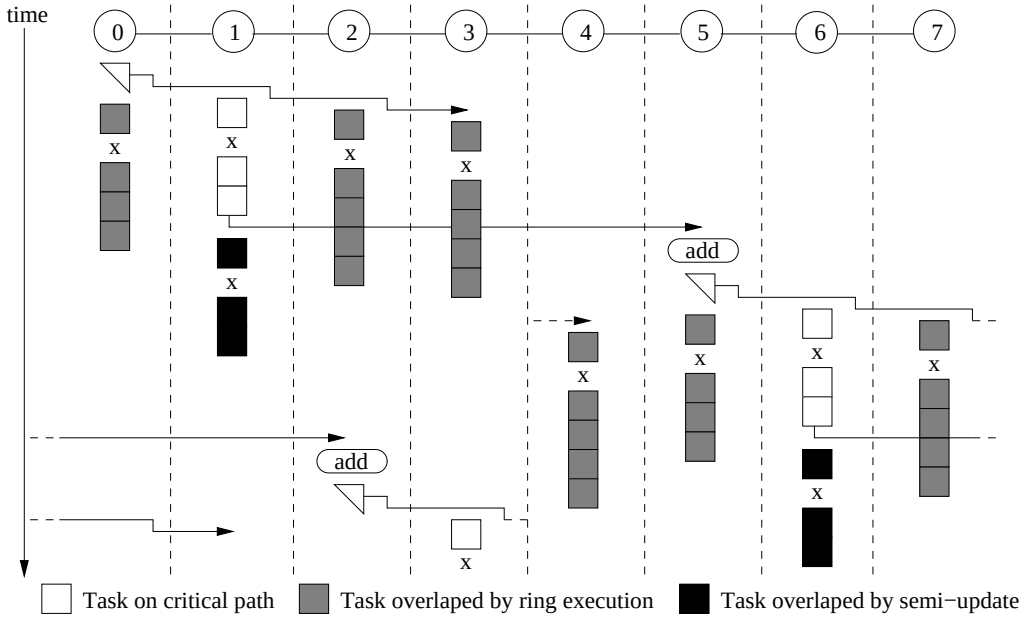


FIG. 4.7. *Execution on a* 2×4 *processor grid of the* `pbdtrsm` *routine.*

Two operations are still to estimate to complete the general case model: The *send* and the *accumulation* operations. For both of them, we have the same restriction on the number of columns as for the first `dgemm` of the update phase. This leads us to the following expressions

$$T_{send} = \left(R \times min\left((p-1)R, R \left\lceil \frac{N - iR}{qR} \right\rceil \right) \right) \texttt{tau} + \texttt{lambda}, \tag{4.7}$$

and

```
fast_comp_time (host, add_desc, &add_time).
```

Here again we handle idle times by applying the same kind of correction, C_u to both sender and receiver of the send operation. Moreover, it has to be noticed that when the number of columns is equal to one, the broadcast operation is replaced by a memory copy. Equation 4.8 gives the computation cost models for the `pbdtrsm` routine on a rectangular grid of processors.

$$\sum_{i=1}^{\lceil N/R \rceil} \left(\texttt{trsm\_time} + T_{broadcast} + C_b + \texttt{dgemm\_time\_1} + T_{send} + C_u + \texttt{add\_time} \right). \tag{4.8}$$

4.5. Accuracy of the Triangular Solve Model. We ran the same kind of experiment as for matrix multiplication to test the accuracy of our triangular solve model. Figure 4.8 shows estimations produced by our model. Comparing these results with those of Figure 4.4, we can see that our model allows us to forecast performance evolution with regard to changes in the processor grid shape. The average error rate is less than 12%. It has to be noticed that if this rate is greater than the one achieved for the multiplication, it comes mostly from the difficulty to model the optimizations done using pipelining. Our model is thus very inaccurate for the 2×4 case, as the execution time is underestimated. However, if we only consider the execution on a processor row, which is the best case, the error rate is then less than 5%.

5. Utility in a Scheduling Context. The objective of our extension of FAST is to provide accurate information allowing a client application, *e.g.,* a scheduler, to determine which is the best solution among several scenari. Let us assume we have two matrices A and B we aim to multiply. These matrices have same

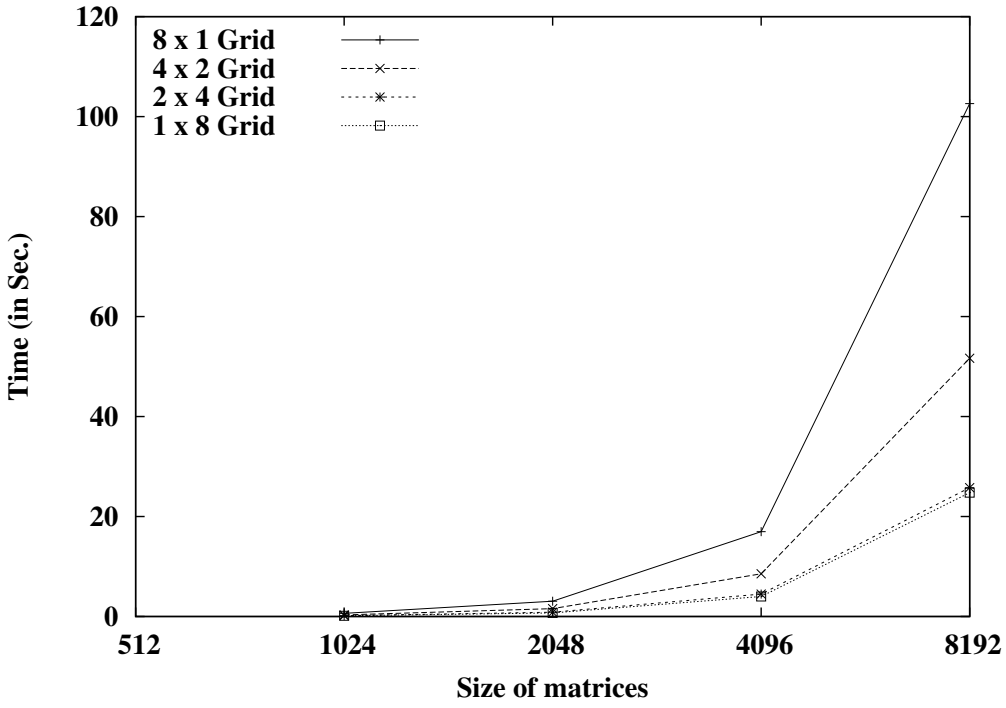


FIG. 4.8. *Estimations of the execution time of a* `pdtrsm` *on different grid shapes.*

size but distributed in a block-cyclic way on two disjointed processor grids (respectively G_a and G_b). In such a case it is mandatory to align matrices before performing the product. Several choices are then possible: Redistribute B on G_a, redistribute A on G_b or define a new virtual grid with all available processors. Figure 5.1 summarizes the framework of this experiment. These grids are actually sets of nodes from a single parallel computer (or cluster). Processors are then homogeneous. Furthermore, inter– and intra–grids communication costs can be considered as similar.

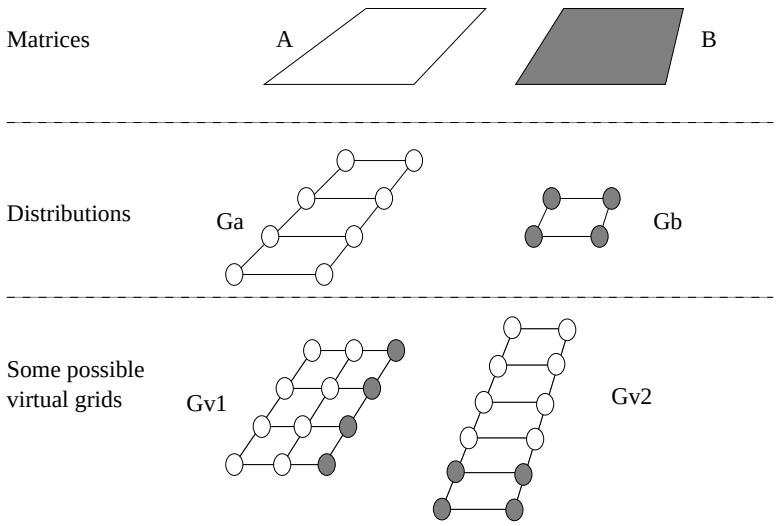


FIG. 5.1. *Initial distribution and processors grids used in this experiment.*

Unfortunately the current version of FAST is not able to estimate the cost of a redistribution between two processor sets. This problem is indeed very hard in the general case [4]. So for this experiment we

have determined amounts of data transfered between each pair of processors and the communication scheme generated by the ScaLAPACK redistribution routine. Then we use FAST to forecast the costs of each point to point communication. Figure 5.2 gives a comparison between forecasted and measured times for each of the grids presented in Figure 5.1.

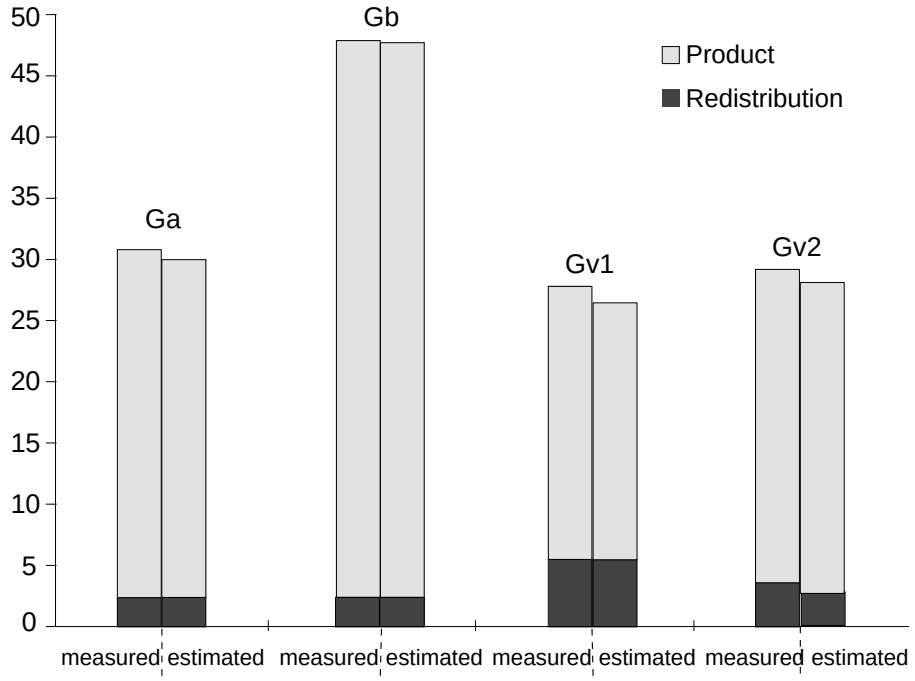


FIG. 5.2. *Validation of the extension in the case of a matrix alignment followed by a multiplication. Forecasted times are compared to measured ones distinguishing redistribution and computation (matrix size* 2000×2000*).*

We can see that the parallel extension of FAST allows to accurately forecast what is the best solution, namely a 4×3 processor grid. If this solution is the most interesting with regards to the computation point of view, it is also the less efficient from the redistribution point of view. The use of FAST can then allow to perform a first selection depending on the processor speed/network bandwidth ratio. Furthermore, it is interesting to see that even if the choice to compute on G_a is a little more expensive, it induces less communications and releases 4 processors for other potential pending tasks. Finally a tool like the extended version of FAST can detect when a computation will need more memory than the available amount of a certain configuration and thus induce swap. Typically the 2×2 processor grid will no longer be considered as soon as we reach a problem size exceeding the total capacity of involved processors. For larger problem sizes the 4×2 grid may also be discarded.

This experiment shows that the extension of FAST to handle parallel routines will be very useful to a scheduler as it provides enough informations to be able to choose according to several criteria: Minimum Completion Time, communication minimization, number of processors involved, ...

6. Conclusion and Future Work. In this paper we proposed an extension to the dynamic performance forecasting tool FAST to handle parallel routines. This extension uses the information provided by the actual version of FAST about sequential routines and network availability. This information is injected into a model coming from code analysis. The result can be considered as a new function it is possible to estimate by call to the FAST API. For instance it will be possible to ask FAST to forecast the execution time of parallel matrix–matrix multiplication or parallel triangular solve.

Some experiments validated the accuracy of the parallel extension either for different grid shapes with a fixed matrix size or for different sizes of matrices on a fixed processor grid. In both cases, the average error rate between estimated and measured execution times is under 4%.

We also showed how a scheduler could benefit of our extension to FAST. Indeed it allows a scheduler to make mapping choices based on a realistic view of the execution platform and accurate estimations for execution

times of tasks and data movement costs.

Our first work will be to extend the work presented in this paper to the entire ScaLAPACK library in order to provide performance forecasting tool for a complete dense linear algebra kernel. Another point to develop is the cost estimation of redistribution. If the general case is a difficult problem, we think we can base our estimations upon a set of redistribution classes. These classes are built depending on modifications made to the source grid to obtain the destination grid. For instance the redistribution from G_a and G_b to G_{v2} made in Section 5 might be elements of the same class, where redistributions only use a proportional increase of the number of rows of the processor grid.

Acknowledgements. This work was supported in part by the projects ACI GRID–GRID ASP and RNTL GASP funded by the French Ministery of research.

We would like to thank the ID laboratory for granting access to its Cluster Computing Center, this work having been done on the ID/HP cluster (`http://icluster.imag.fr/`).

REFERENCES

[1] J. Bourgeois, F. Spies, and M. Trhel, *Performance Prediction of Distributed Applications Running on Network of Workstations*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), H. R. Arabnia, ed., vol. II, Las Vegas, June 1999, CSREA Press, pp. 672–678.

[2] E. Caron, D. Lazure, and G. Utard, *Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization*, in Proceedings of the 7th International Conference on High Performance Computing (HiPC'00), vol. 1593 of Lecture Notes in Computer Science, Springer-Verlag, Dec. 2000, pp. 161–172.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: A Practical Model of Parallel Computation*, Communications of the ACM, 39 (1996), pp. 78–95.

[4] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert, *Scheduling Block-Cyclic Array Redistribution*, in Parallel Computing: Fundamentals, Applications and New Directions, E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg, eds., North Holland, 1998, pp. 227–234.

[5] F. Desprez, M. Quinson, and F. Suter, *Dynamic Performance Forecasting for Network-Enabled Servers in a Heterogeneous Environment*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), H. Arabnia, ed., vol. III, Las Vegas, June 2001, CSREA Press, pp. 1421–1427. ISBN: 1-892512-69-6.

[6] S. Domas, F. Desprez, and B. Tourancheau, *Optimization of the ScaLAPACK LU Factorization Routine Using Communication/Computation Overlap*, in Proceedings of Europar'96 Parallel Processing Conference, vol. 1124 of Lecture Notes in Computer Science, Springer Verlag, Aug. 1996, pp. 3–10.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An Extended Set of Fortran Basic Linear Algebra Subroutines*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–17.

[8] J. Dongarra, A. Petitet, and R. C. Whaley, *Automated Empirical Optimizations of Software and the ATLAS Project*, Parallel Computing, 27 (2001), pp. 3–35.

[9] J. Dongarra and K. Roche, *Deploying Parallel Numerical Library Routines to Cluster Computing in a Self Adapting Fashion*, Submitted to Parallel Computing, (2002).

[10] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann, 1998. ISBN 1-55860-475-8.

[11] T. Howes, M. Smith, and G. Good, *Understanding and Deploying LDAP Directory Services*, Macmillian Technical Publishing, 1999. ISBN: 1-57870-070-1.

[12] M. Quinson, *Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment*, in Proceedings of the International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02), Fort Lauderdale, Apr. 2002.

[13] V. Rayward-Smith, *UET Scheduling with Unit Interprocessor Communication Delays*, Discrete Applied Mathematics, 18 (1987), pp. 55–71.

[14] R. Wolski, N. Spring, and J. Hayes, *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, Future Generation Computing Systems, Metacomputing Issue, 15 (1999), pp. 757–768.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: October 01, 2002
Accepted: December 21, 2002

Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 6, Number 1, pp. 71–80. http://www.scpe.org

PROBES COORDINATION PROTOCOL FOR NETWORK PERFORMANCE MEASUREMENT IN GRID COMPUTING ENVIRONMENT

R. HARAKALY*, P. PRIMET†, F. BONNASSIEUX‡, AND B. GAIDIOZ§

Abstract. The fast expansion of Grid technologies emphasizes the importance of network performance measurement. Some network measurement methods, like TCP throughput or latency evaluation, are very sensitive to concurrent measurements that may devalue the results. This paper presents the Probes Coordination Protocol (PCP) which can be used to schedule different network monitoring tasks. In addition, this paper goes on to discuss the main properties of the protocol; these being, flexibility, efficiency, robustness, scalability and security. This study presents the results of its evaluation and of experiment periodicity measurements.

Introduction. The purpose of Computational Grids is to aggregate large collections of shared resources (computing, communication, storage, information) in order to build an effective and high performance computing environment for data-intensive or computing-intensive applications. The underlying communication infrastructure of these large scale distributed environments consists of a complex interconnection of the public Internet, local area networks and high performance system area networks like Myrinet. Consequently "the network cloud" may exhibit extreme heterogeneity in performance and reliability that can considerably effect the distributed application performance. In a Grid environment, monitoring the network is, therefore, critical in determining the source of performance problems or in fine tuning the system to perform better. For such purposes a network performance measurement system may be deployed over the Grid and net cost function may be computed and provided to a grid resource allocation component. Sensors that aim to measure the different network metrics such as end to end throughput, loss rate or end to end delay are the basic building blocks of the network performance measurement system. However, classical Internet measurement tools can also be used for this purpose. Two kinds of measurement methodology are classically applied (these being active and passive methods).

Active methods inject extra traffic to determine the capacity of the links in terms of latency, loss or bandwidth. Passive methods measure the traffic but are unable to evaluate the real capacity of a link. For example to evaluate the available TCP or UDP throughput tools like Iperf [1] or Netperf [2] send probe packets during a given duration (default $10s$). Amounts of send probe data are from 12.5 MB on a 10Mbps link to 1.25 GB on the 1Gbps link.

As the traffic generated by active testing is added to the usual traffic load on the network, there are drawbacks to active methodologies. Firstly, they add a potentially burdensome load to the network; secondly, the additional traffic may perturb the network and devalue the resulting analysis. Therefore, these tools must be appropriately scheduled to minimize the impact on networks while still providing an accurate measurement of a particular network metric.

Grid network monitoring raises problems which are not so critical in classical Internet performance measurement. As the number of sites and their respective logical links used by a community of users in a Grid environment are finite the probability of concurrent measurements is high.

The possibility of sensor probes colliding and thereby measuring the effect of sensor traffic increases quadratically with the number of sensors [3]. This can become highly critical in hierarchical Grids like the HEP physics DataGrid (see section 3), organized following a multi-tiered architecture. For example, the probes from tier 1 to tier 0 (CERN) may collide frequently making tier 0 site a real bottleneck (see fig. 0.1). This leads to chaotic results.

In this paper the main features of the protocol which we have developed for coordinating the network monitoring probes in the European Data Grid (EDG) project [5], are described.

The paper is organized as follows. In the first section probes coordination service and four possible experiment scheduling strategies as well, as requirements which must be fulfilled for probes coordination service, are discussed. The second section goes on to describe probes coordination protocol, its design principle and characteristics. In a third section our implementation of PCP, with EDG distribution, the evaluation methodology and the results of testing are presented. Finally related work is discussed.

*CNRS-UREC, ENS-LIP, 46, allée d'Italie, 69 364 Lyon, France (robert.harakaly@ens-lyon.fr)
†INRIA-RESO, ENS, 46, allée d'Italie, 69 364 Lyon, France (pascale.primet@ens-lyon.fr)
‡CNRS-UREC, ENS-LIP, 46, allée d'Italie, 69 364 Lyon, France (franck.bonnassieux@ens-lyon.fr)
§INRIA-RESO, ENS, 46, allée d'Italie, 69 364 Lyon, France (benjamin.gaidioz@ens-lyon.fr)

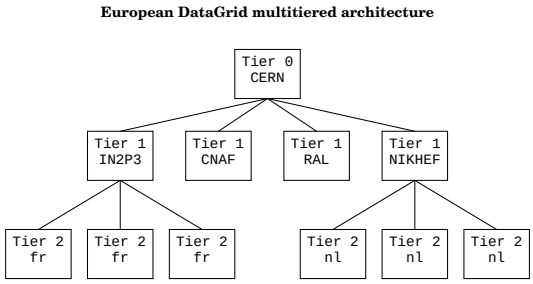


FIG. 0.1. *Multi-tiered architecture of the European DataGrid project.*

1. Probes coordination service.

1.1. Strategies. For coordinating active probes, different approaches, from very optimistic to very pessimistic are possible. Optimistic strategy, considers that the probability of measurements collision is relatively low. At the opposite, pessimistic strategy aims to avoid any measurement collision. Four main approaches are possible:

- random scheduling
- cron based distributed scheduling
- centralized scheduling
- token passing

1.1.1. Random scheduling. The most optimistic and simple method is random activation of the sensors. We define this as random scheduling. This type of strategy assumes that load generated by active measurement is negligible and considers that all traffic is production traffic. This method is valid for non-systematic monitoring.

1.1.2. Cron based distributed scheduling. The most popular scheduling method uses *cron* daemon scheduler on each measurement host. Good time synchronization between hosts is required. The smallest time slot enabled by the cron mechanism is one minute. Any time shift, or unexpected long measurement duration can cause collision. This strategy is valuable only if the number of sensors is small or if the measurements are non-intrusive.

1.1.3. Centralized scheduling. At the midpoint between the optimistic and pessimistic scheduling strategies lies the "measurement on demand" strategy. Here a central server coordinates the experiments. This strategy has a single point of failure and is not scalable.

1.1.4. Token passing. Considering high collision probability in a grid context one has to adopt a pessimistic strategy to avoid contention and to provide a scalable way to generate pertinent network performance measurement. To realize such pessimistic strategy, a token passing protocol may be used (see fig. 1.1).

1.2. Requirements. In a grid context, and especially in the case of a hierarchical ones, the pessimistic strategy has to be privileged. A mechanism implementing such synchronization strategy has to exhibit the traditional properties of a grid service, that are: flexibility, efficiency, robustness, scalability and security.

2. Probes Coordination Protocol.

2.1. Design principle. Probes coordination protocol is a standalone protocol dedicated to network monitoring. It is based on a token passing protocol and on a clique concept inspired from the NWS approach [3]. PCP is open in the sense that it can support any type of sensor. It implements many original features like distributed registry, inter-clique synchronization and security.

2.1.1. Clique definition. The basic building block of PCP is the concept of clique, logical group of sensors. It is defined by an ordered list of participating nodes to which type of sensor and additional required or optional information like period, delay and timeout are attached. Figure 2.1 gives an example of clique definition. The clique definition is then converted into a token. Each token is registered in a distributed registry. The distributed registration process lies on a peer-to-peer approach. Each node belonging to any clique maintains all necessary information concerning the PCP protocol in its local registry. Thanks to that the maximum tolerance of network or host failure was achieved.

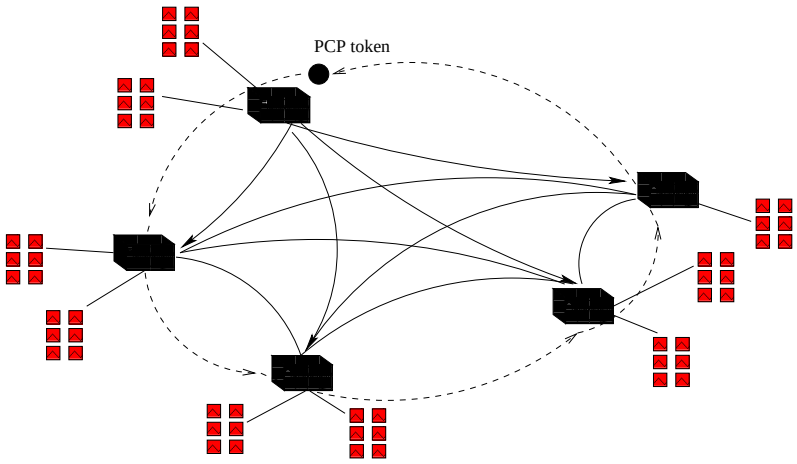


FIG. 1.1. *Probes scheduling using token passing protocol.*

```
1)
  name:date
  member:wp7.ens-lyon.fr
  member:netmon.grid.science.upjs.sk
  member:ccwp7.in2p3.fr
  period:120
  timeout:10
  delay:4
  command:iperf -c netmon.cern.ch >> netmon.cern.ch.log

or
2)
  command:iperf -c %m >> %n.log
```

FIG. 2.1. *Sample PCP clique definition. 1) Three member sites measure available throughput achievable on their link to CERN using iperf tool. 2) parameters %m and %n will be replaced by member host IP address and name, respectively. It enables to create full mesh measurements easily. In both cases the results (stdout) are appended to the <hostname>.log files.*

2.1.2. Token structure. Token is a data structure exchanged within the PCP protocol. Three token types are defined. Different types of tokens are identified by *type_of_token* field in the token header (see Table 2.1) and they are processed according their type.

Measurement token is the main token type. It is used for periodic activation of the sensors. Contains information from clique definition combined with the security information to authenticate the token's owner as well as the integrity of the token information. Table 2.1 illustrates the measurement token structure. The fields can be grouped into different groups following defined characteristics of the protocol. *Type_of_token* field distinguish different types of the token. Main definition of the clique is in the fields *clique_name* which is the unique name of the clique, list of members saved in the array *members[]*, field *periodicity* and *shell_command* defining the sensor. Field *timedelay* belongs to the group of scheduling accuracy attributes. Protocol robustness is ensured by set of fields *token_id*, *regeneration_host* and *timeout* and security by fields *owner_s_email* and *crypted_signature*. Field *option* is used to distribute additional information. These fields will be discussed in the next sections of the paper.

The measurement token processing algorithm is illustrated in the fig. 2.2. After the token identification phase the security checks (see section 2.4) are performed. After success of the security procedure, the local token registry information is updated and the clique action procedure starts. This procedure consists of experiment time adjustment (see section 2.3), optional host locking and performing the defined clique action and optional site locks are released. Finally if the token is newly registered on the site, an optional delay of *timedelay* seconds is inserted and then the token is sent to the next hop.

TABLE 2.1
Structure of the token

```
{
    byte type_of_token

    string clique_name
    host_id members[]
    time periodicity

    string shell_command

    time timedelay

    integer token_id
    host_id regeneration_host
    time timeout

    byte option

    string owner_s_email
    string crypted_signature
}
```

For distribution of control messages between PCP nodes, two additional token types were defined.
Distributed control token these tokens implements clique broadcasting functions when the information has to be delivered to each member site. Typical example of this is a token_stop function.
Simple control token implements simple functions where information is exchanged between two nodes. There are token_start and token_update functions and host_locking tokens.

2.2. Characteristics.

2.2.1. Flexibility. It is one of the original characteristic of PCP. It was designed to be flexible in terms of definition of the action and its customizability. Protocol is not bound to any measurement system or tool. The *shell_command* parameter allows to fully customize the actions.

2.2.2. Protocol robustness. The robustness is very important for such a distributed protocol. Monitoring data and services using these data like forecasting and network cost function needs to be continuously available. This impose the very strong requirement on the monitoring longevity. To reach high robustness and fault tolerance the protocol is based on a peer-to-peer approach, which insures high independence of participating sites. Nevertheless the token passing architecture, raises several classical important issues that have to be addressed.
Lost token: The token might be lost due to host or network failures. Token timeout based token regeneration mechanism initiates a creation of new token after timeout expiration.
Delayed token: In case of big instabilities of the network it may happen that measurement takes longer than typically. The token will not be received in time at the next hop and in some cases the timeout will expire. In such situation the token regeneration mechanism will initiate a new token, like in the case of the lost token. After end of the long experiment, the original token will be sent to this hop. Since this host has already regenerated a new token, this original token will be discarded. Newly generated token and original ones are differentiated by using the *token_id* field which is incremented at each regeneration. Thus, if a system receives token with *token_id* lower than is locally registered, it will discard a token.
Token storm: Occurs when tokens with the same *token_id* on subsequent hosts are created. It might happen in case of misconfiguration of the system, or in case of real network instabilities. To solve this issue the *regenerating_host* identifier was used, which characterizes the site which has regenerated the token. If a system receives token with *regenerating_host* lower than is locally registered it will discard it.

Figure 2.3 displays four scenarios of token passing.
a) shows typical scenario when token is delivered in time.
b) Lost token: Token is lost between node 2 and node 3, that will, after expiration of the timeout, lead to token regeneration on the node 3.

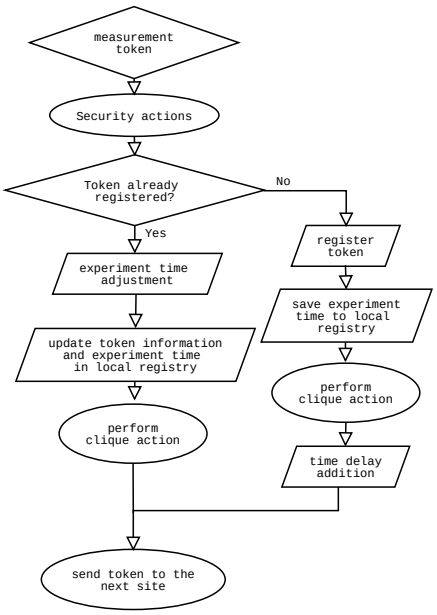


FIG. 2.2. *PCP experiment token passing through system*

c) Delayed token: Due to network instabilities the measurement on the node 2 takes very long. It causes expiration of the timeout on the node 3, which regenerates a token and start measurement. The former token is delivered after the timeout expiration to the node 3 which discards it.
d) Token storm: Creation of the token storm due to small timeout parameter value. As in the previous case, network instability causes long measurement delay on node 2. Node 3 regenerates a token, but due to shift of experiment start time, the experiment on node 3 cannot be finished before timeout expiration on a node 4. This regenerates again the token. The delayed tokens from node 2 and node 3 the receiving nodes will be discarded.

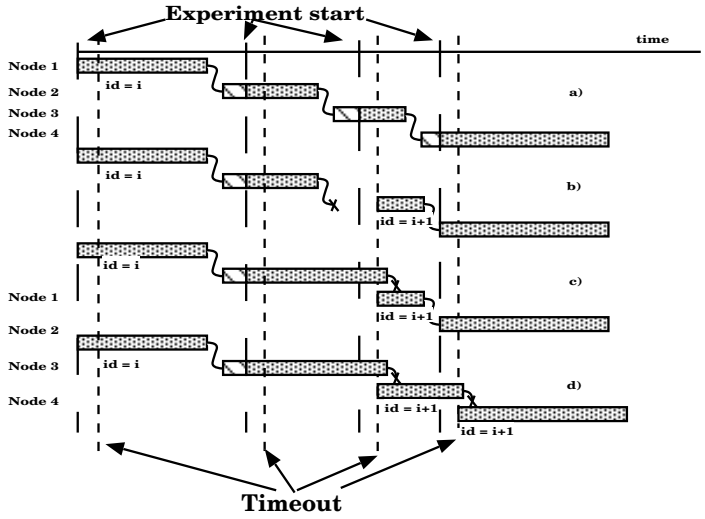


FIG. 2.3. *Token passing issues. a) normal passing, b) lost token, c) delayed token and d) token storm.*

2.2.3. Scalability. The distributed nature of the protocol makes it very scalable. Distributed registration gives high scalability. Each participating node maintain all necessary information in its local registry, as it is illustrated on figure 2.4. In this figure two cliques are defined: "pinger clique" and "iperf clique". Each node

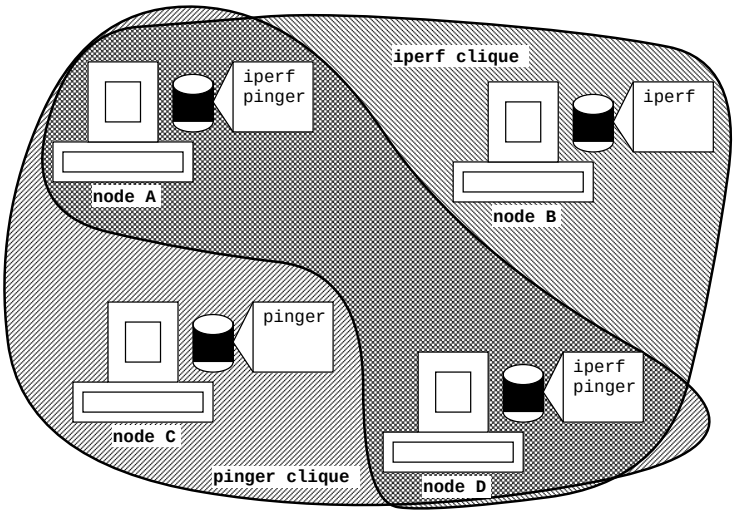


FIG. 2.4. *PCP distributed registry architecture. Each node creates its own local registry of cliques/tokens it belongs.*

registers only cliques it belongs to (nodes A, C and D registers clique pinger and nodes A, B and D registers clique iperf).

A large number of sensors can be easily managed by grouping them in multiple logical groups. They might be defined according to the network topology or any other characteristics of the measured system. In the case of the EDG project the cliques definition policy reflects the multi-tiered architecture.

Since measurement host might run multiple sensors as well as any sensor might be member of multiple cliques, it is necessary to analyze the interactions between different cliques. For example the Tier1 IN2P3 site in the fig. 0.1 belongs to two cliques. If the probe is bandwidth measurement like Iperf, the Tier1 site might be simultaneously source/destination of two experiments of two cliques, making the measurement unreliable. To solve this problem an inter-clique synchronization mechanism based on host locking has been introduced. Both source and destination hosts are locked during experiment. If other lock request arrives to one of these nodes, the requesting node will be blocked until the lock is released.

Figure 2.5 illustrates the host locking mechanism. In this figure *node* 1 to *node* 3 are members of the Clique 1 and *node* 3 and *node* 4 are members to the Clique 2. At time A *node* 1 sends locking request to *node* 2 (*message* 1), it is locked and returns successful result in message 2. After *node* 1 starts monitoring experiment (message 3). At the end of the experiment information message 4 is sent from *node* 2 to *node* 1. *Node* 1 asks to unlock the *node* 2 (message 5) which answers with the unlocking status (time B, message 6) and starts its own measurement to the *node* 3 with the locking request (message 1). Since *node* 3 in time C is still locked due to the experiment with *node* 4, *node* 3 answers to the locking request with the error message ('!'). It causes the *node* 2 to wait for the end of the experiment with the *node* 3. *Node* 3 adds the *node* 2 to the waiting list and after successful unlocking it informs the *node* 2 with the information message 7 that it is unlocked. This message is followed by locking request from *host* 2.

Host locking mechanism raises an issue of the deadlocks. Deadlocks may strongly disturb the measurement and destroy the measurement periodicity. The detailed study of the measurement scenario and good clique creation policy are necessary to minimize the probability of deadlock creation. To enable the system to release itself from a eventual deadlock state, we introduce the timeout mechanism that releases the locks and prevent deadlocks.

2.3. Scheduling accuracy. Network monitoring data can be used for many different purposes. They may serve as information for network administrators as well as for optimization of data transfers and resource scheduling in grid systems. The network monitoring data may also serve as a basis for prediction of network metrics. Currently used prediction methods (e.g. [3]) require good measurement periodicity. To meet this requirement PCP has a built-in mechanism for start time adjustment.

Two main factors influences the experiment periodicity. First is network experiment length variation and

second is the fluctuation of the token transmission time. Both are due to the network instable behavior.

The aim of start time adjustment is to achieve highest timing precision possible. Experiment is triggered using time-stamp saved into local registry during previous experiment and the periodicity value. If the token arrives in advance, the start of clique action is delayed, otherwise the clique action is executed immediately and experiment execution time is adjusted.

$$t_w = (t_{n-1} + p) - t$$

where t_w is time adjustment delay, t_{n-1} is the experiment time-stamp saved during previous experiment, p is the periodicity and t is current time.

In order to adapt to the experiment length and transmission delay fluctuations we introduced an optional parameter called *timedelay*. This timedelay is inserted after the experiment only during the first round (fig. 2.6). It, in conjunction with start time adjustment enables to absorb most of mentioned fluctuations and to achieve good measurement periodicity even at the beginning of the measurement.

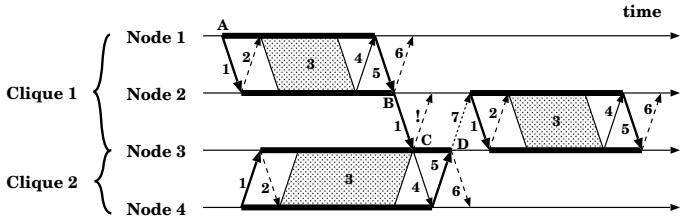


FIG. 2.5. *Communication using inter-clique locking protocol.*

2.4. Security. In distributed computing the security is one of the most important issue. Although PCP is designed like user level process without special privileges, the possibility to launch any external program requires to provide the highest security level possible. Security issues might be divided into two groups:

- **Host security:** Local system administrator must have full control over the host. There must be possibility to define local security policies, authentication of the token owners, authorization of the tools used. Token owner's authorization is implemented using its digital signature and its validation using the public key of authorized person, and tool authorization is done by copying it to the `$PCPD_HOME/scripts` directory.
- **Token security:** In this part the goal is to provide and ensure access rights to the token information and to the token itself. It is necessary to ensure, that content of the token might be changed only by its owner and in case of the violation, to detect it. In case of the token function (*token_stop* or *token_update*) enable it only in case it is issued/signed by token owner.

Previous items shows the necessity of assignment the owner to the token and introduction of the security fields into the token structure. Table 2.1 shows the token structure with the security fields: token owner's e-mail, and its digital signature. Token is signed using crucial token information (cannot be changed using token_update function) as token name, clique action and token owner's e-mail and owner's private key. The signature is included into the signature field of the token.

3. Application and results.

3.1. European DataGrid project. The PCP protocol has been developed within the European DataGrid project that aims to develop, implement and exploit a large-scale data and CPU-oriented computational GRID [5]. The objective of this project is to enable next generation scientific exploration that requires intensive computation and analysis of shared large-scale databases, from hundreds of TeraBytes to PetaBytes, across widely distributed scientific communities. The EDG testbed comprises today more than 40 sites with hundreds of processors organized in a multi-tiered architecture.

3.2. Grid Network Monitoring. To optimize the usage of the networks a dedicated network monitoring system has been developed and is under deployment on this testbed. The Grid network aware applications or the middleware components, dedicated to the resource usage optimization in the Grid, such as a resource broker, a job scheduler or a replica manager are using the monitoring data to adjust their behavior to make the

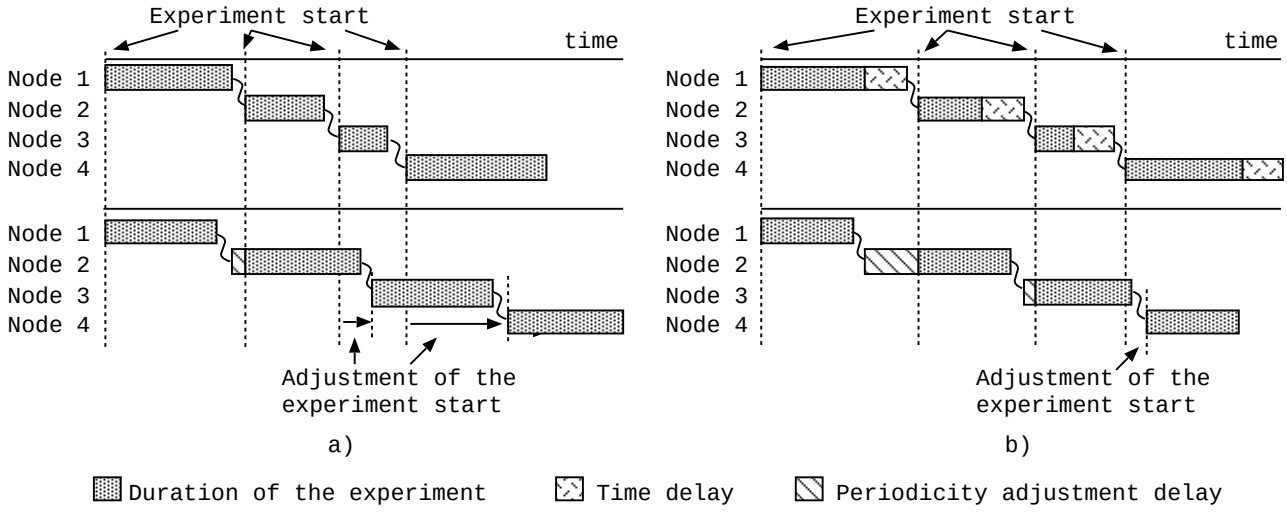


FIG. 2.6. *Token passing in first and second round without time delay a) and with time delay b).*

best use of this resource. A **network cost function** that is able to compare two destinations is provided. It is computed from collected metrics. For example, given src a source site, $dest_1$ and $dest_2$, two destination sites, v an amount of data to be transferred, r_1 et r_2 the estimated *throughput* of link1 and link2. The network cost function f is expressed as a transfer delay that is the relevant metric for the optimizer. We have:

$$(f(src, dest_1, v) = v/r_1) and (f(src, dest_2, v) = v/r_2)$$

and

$$(f(src, dest_1, v) \leq f(src, dest_2, v)) or (f(src, dest_2, v) \leq f(src, dest_1, v))$$

EDG GridNetMon is a prototype set of tools for network monitoring within the framework of a simple and extensible architecture that permits the basic network metrics to be published to the Grid middleware, and also for them to be available, via visualization, to the human observer. The architecture for network monitoring comprises four functional units, namely, monitoring tools or sensors, a repository for collected data, the means of analysis of that data to generate network metrics and the means to access and to use the derived metrics. The EDG Network monitoring system implements basic monitoring tools which produce the standard measurements: Round Trip Delay, Packet Loss, Total Traffic volume, TCP and UDP Throughput, Site Connectivity and Service Availability. Well known monitoring tools have been tested and integrated in the architecture. New specific software has also been developed within the project to fill several gaps.

Grid network measurement architecture raise lot of specific problems like deciding where, when and how to deploy the sensors in the Grid, but also deciding how to schedule the measurements especially when they may be intrusive. PCP protocol address this last issue.

We have implemented PCP under RedHat 7.3 distribution of the Linux in a daemon called *pcpd*. To support multiple tokens the multi-threaded architecture is used. Current version of the pcpd implements basic features of PCP. The inter-clique scheduling is actually not supported. It implements only authorization of allowed commands/scripts by administrator by installing them to `$PCPD_HOME/scripts` directory.

3.3. Evaluation. We set up a methodology which consists of different actions to evaluate the protocol. To test individual properties of the protocol we built up different testbeds with different network properties.

- **Local area testbed,** this type of network provides low latency with negligible performance fluctuations and zero packet loss. This simplest testbed is used for testing of the basic functionalities of the protocol and its flexibility.

- **Well provisioned testbed,** this testbed provides higher latency between $10ms$ and $30ms$, small performance fluctuations and low packet losses. These testbeds are used to study the scalability issues and the inter-clique synchronization.
- **Wide area unstable testbed,** this platform provides latencies up to $80ms$ with large instabilities and important packet loss. This testbed serves for the validation of the protocol robustness (the timeout based token regeneration as well as elimination of the duplicate tokens or token storms), scalability and the scheduling accuracy (influence of the *timedelay* parameter to the scheduling accuracy and the start time adjustment).

These testbeds are built on the network of ENS Lyon, metropolitan network of Lyon, EDG WP7 testbed based on European academic core network GEANT and national NRNs. WAN testbed adds hosts in Slovakia.

Here are presented the results of the evaluations in the WAN testbed. Network connection during the evaluation process showed instabilities due to traffic in the bottleneck connection between Slovak NRN (SANET), and GEANT network. Thanks to that the protocol robustness and scheduling accuracy were studied. Long-term tests were conducted for approximately $2-3$ weeks. Results prove high resistivity of the protocol against network instabilities and effectivity of the timeout-based token regeneration mechanism. Since network connection showed important fluctuations of the network parameters (latency, available bandwidth) we studied the scheduling accuracy of the protocol. Experiment was done using one clique consisting of four member sites (two in France and two in Slovakia). Periodicity of the measurement was set to $120s$, timeout parameter was set to $10s$ and timedelay parameter was set to $4s$. Results of this measurement are displayed on Figure 3.1. It shows the histogram of measured periodicities. The highest peak corresponds to the periodicity of $120s$. Satellite peaks (emphasized in the inset) corresponds to the experiment periodicity dispersion due to $1s$ resolution of the used timer and peak at $130s$ corresponds to activation of token regeneration mechanism.

Current version of pcpd is deployed on some EDG monitoring testbed sites to schedule intrusive Iperf-based available throughput measurements. During future deployment of the PCP, the scalability and inter-clique scheduling will be studied.

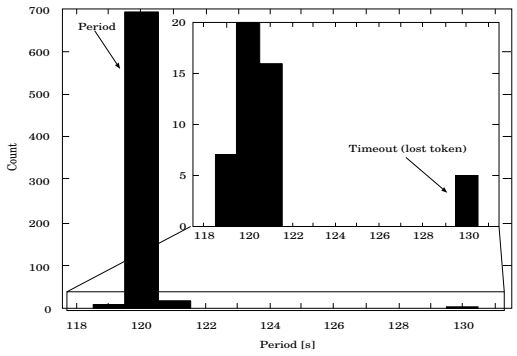


FIG. 3.1. *Measured experiment periodicity distribution. Experiment period* $120s$*, timeout* $10s$*. Peak at* $130s$ *is due to timeout expiration and token regeneration*

4. Related work. As presented in section 1.1 different scenarios are possible to schedule network monitoring activities. Appropriate scenario depends on the used tool as well as on a measurement architecture.

Cron based distributed scheduling approach is adopted for example in PingER tool [6]. Ping experiments are nonintrusive thus the optimistic scenario is valid.

Another example of using a cron based scheduling is a IperER tool. Iperf measurements may be very intrusive. That makes this approach quite limited. It request very good time synchronization on the measurement nodes and a central (administrative) agreement about the reservation of the timeslots for each measurement node to avoid conflicting measurements. The minimum of 1 minute timeslot per experiment, given by cron daemon, strongly reduces the scalability of the mechanism.

RTPL (Remote Throughout Ping Load) tool [7] adopts centralized scheduling scenario. Intrusive available throughput experiments are scheduled by a central server, which initiate execution on remote node. This approach is limited to small or centralized experiments. As it uses central server it has a single point of failure.

Network Weather System (NWS) [3] uses token passing approach. It defines a *Clique protocol* to schedule exclusive network measurements. Clique protocol is embedded in the NWS infrastructure and it is fully dedicated to built-in monitoring actions and don't support the scheduling of external sensors.

5. Conclusion. In this article we presented Probe Coordination Protocol. It is based on the pessimistic scheduling scenario. This protocol is open, scalable, highly customizable, secure and robust.
We present an evaluation of our implementation of the PCP protocol. It is deployed within the European DataGrid project on several sites for synchronizing the network experiments executed by EDG grid monitoring infrastructure. We observed very good robustness and fault tolerance of the protocol and efficiency of the timeout based token regeneration. We reported the results of experiment periodicity measurements. They showed good periodicity distribution.

This lightweight and robust peer-to-peer tool may be adapted to implement any type of synchronization or ordered broadcast in grid environment.

Acknowledgments. This work is supported by European DataGrid project IST-2000-25182, Unité Réseaux du CNRS and INRIA project RESO.

REFERENCES

[1] Iperf Home Page: `http://dast.nlanr.net/Projects/Iperf`
[2] Netperf Home Page: `http://www.netperf.org/`
[3] R.Wolski, N.T.Spring, J. Hayes, *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing,* Future Generation Computer Systems, 1998
[4] B. Gaidioz, R. Wolski and B. Tourancheau, *Synchronizing network probes to avoid measurement intrusiveness with the network weather service,* In Proc. 9th IEEE Symp. on High Performance Distributed Computing, 2000
[5] DataGrid project Home Page: `http://www.eu-datagrid.org/`
[6] L. Cottrell, W. Matthews, C. Logg, *Tutorial on Internet monitoring and pinger at slac*, July 1999, Available from `http://www-iepm.slac.stanford.edu/comp/net/wanmon/tutorial.html`
[7] RTPL Home Page: `http://fseven.phys.uu.nl/~blom/rtpl/index.html`

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: September 30, 2002
Accepted: December 09, 2002

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 81–98. http://www.scpe.org
ISSN 1895-1767

SERIALIZATION OF DISTRIBUTED THREADS IN JAVA

DANNY WEYNS, EDDY TRUYEN, PIERRE VERBAETEN*

Abstract. In this paper we present a mechanism for serializing the execution-state of a distributed Java application that is implemented on a conventional Object Request Broker (ORB) architecture such as Java Remote Method Invocation (RMI). To support serialization of distributed execution-state, we developed a byte code transformer and associated management subsystem that adds this functionality to a Java application by extracting execution-state from the application code. An important benefit of our mechanism is its portability. It can transparently be integrated into any legacy Java application. Furthermore, it does require no modifications to the Java Virtual Machine (JVM) or to the underlying ORB. Our serialization mechanism can serve many purposes such as migrating execution-state over the network or storing it on disk. In particular, we describe the implementation of a prototype for repartitioning distributed Java applications at run-time. Proper partitioning of distributed objects over the different machines is critical to the global performance of the distributed application. Methods for partitioning exist, and employ a graph-based model of the application being partitioned. Our mechanism enables then applying these methods at any point in an ongoing distributed computation. In the implementation of the management subsystem, we experienced the problem of losing logical thread identity when the distributed control flow crosses address space boundaries. We solved this well known problem by introducing the generic notion of distributed thread identity in Java programming. Propagation of a globally unique, distributed thread identity provides a uniform mechanism by which all the program's constituent objects involved in a distributed control flow can uniquely refer to that distributed thread as one and the same computational entity.

Key words. serialization of execution-state, distributed threads, Java

1. Introduction. In this paper we present a mechanism for serializing the exe-cution–state of a distributed Java application. We describe this mechanism in the context of a system for run-time repartitioning of distributed Java applications. For distributed object–oriented applications, an important management aspect is the partitioning of objects such that workload is equally spread over the available machines and network communication is minimized. Traditional techniques for automatic partitioning of distributed object applications uses graph–based algorithms (e.g. [8]).

In a static approach an external monitor automatically determines the best possible partitioning of the application, based on observation of behavior of the application (i.e., the dispersal of costs) during a number of representative runs. This partitioning is fixed for the entire execution of the application. However in a dynamic environment the optimal object distribution may change during execution of the application. To cope with this, the external monitor may periodically check the workload at run-time on each separate machine. Whenever the workload on one or more machines crosses a certain threshold, e.g., following the low–water high–water workload model as described in [12], the monitor immediately triggers the repartitioning algorithm and relocates one or more objects to another machine.

The relocation of a running object involves the migration of its object code, data state and execution-state. Conventional Java–based Object Request Brokers (ORBs), such as the Voyager ORB [9], support passive object migration (migration of object code and data state, but no migration of execution-state). However, run-time repartitioning does not want to wait with object relocation until that object and eventually all objects involved in the execution of that object are passive. Instead it aims to handle the triggers for object repartitioning immediately. As a consequence existing methods for repartitioning must be adapted to be applied at any point in an ongoing distributed computation. As such, it is necessary to support object relocation with migration of execution-state. Migration of execution-state is in the literature often referred to as strong thread migration [7]. The fact that the Voyager ORB does not support strong thread migration is not just a missing feature, but the real problem is that migration of execution-state is simply not supported by current Java technology.

To solve this we developed a byte code transformer and associated management subsystem that enables an external control instance (such as the above load balancing monitor) to capture and reestablish the execution-state of a running distributed application. We call this serialization of distributed execution-state. The byte code transformer instruments the application code by inserting code blocks that extract the execution-state from the application code. The management subsystem, which is invoked by the inserted codes, is responsible for managing the captured execution-state efficiently. The management subsystem also provides operations by which an external control instance can initiate serialization of the distributed execution-state at its own will.

It is important to know that we solely focus on distributed applications that are developed using conventional

*Department of Computer Science, DistriNet, Katholieke Universiteit Leuven, Belgium
(email: danny-eddy-pv@cs.kuleuven.ac.be; web: www.cs.kuleuven.ac.be/~ danny/DistributedBRAKES.html)

ORBs such as Java Remote Method Invocation (RMI) or Voyager. Programmers often use these middleware platforms because of their object–based Remote Procedure Call (RPC) programming model, which is very similar to the well-known object-oriented programming style.

1.1. Important aspects of our work. In this paper we first describe how we realized **serialization of a distributed execution-state**. Note that in the past, several algorithms have been proposed to capture the execution state of Java Virtual Machine (JVM) threads. Some require the modification of the JVM [3]. Others are based on the modification of source code [7]. Some algorithms rely on byte code rewrite schemes, e.g. [11, 17]. We too had already implemented such a byte code rewrite algorithm called Brakes [15]. However, most of these schemes are presented in the domain of mobile agents systems. Whenever a mobile agent wants to migrate, it initiates the capturing of its own execution-state. As soon as the execution-state is serialized the agent migrates with its serialized execution-state to the target host where execution is resumed. However, serialization of distributed execution-state of Java RMI-like applications introduces two aspects that are not covered in this existing thread capturing systems. First, contrary to how conventional Java threads are confined to a single address space, the computational entities in Java RMI applications execute as distributed flows of control that may cross physical JVM boundaries. In the remainder of this paper we call such distributed flows of control *distributed threads* [4]. Serializing the execution-state of a distributed application boils down to capturing all distributed threads that execute in that application. The second aspect is that mobile agents initiate the capturing/reestablishment of their execution-state themselves, whereas capturing/reestablishment of distributed execution-state must often be initiated by an external control instance. The Brakes thread serialization scheme is not designed for being initiated by such an external control instance. In this paper we describe how we have extended Brakes with (1) a mechanism for serialization of the execution-state of distributed threads that (2) can be initiated by an external control instance.

Subsequently, we show how we used this serialization mechanism to implement a prototype for **run-time repartitioning of distributed Java applications**. The idea is that a load balancing monitor, that plays the role of external control instance here, captures the execution-state of an application whenever it wants to repartitioning that application and reestablishes the execution-state after the repartitioning is finished. The advantage of having separate phases for migration of execution-state and object migration is that objects can migrate independently of their (suspended) activity. Requests for object migration can immediately be performed, without having to wait for the objects to become passive. This is possible because, by using the serialization mechanism, application objects can be turned passive on demand by the monitor, while their actual execution-state is safely stored in the management subsystem of the serialization mechanism. So when the actual repartitioning takes place, all application objects are a priori passive. As a result, we can still use conventional passive object migration support to implement the run-time repartitioning prototype. In this paper we assume that the application's underlying ORB supports passive object migration (e.g., the Voyager ORB), but existing work [6] has shown that support for passive object migration can also be added to the application by means of a byte code transformation.

Previous work [10][16] already offers support for run-time repartitioning, but this is implemented in the form of a new middleware platform with a dedicated execution model and programming model. A disadvantage of this approach is that Java RMI legacy applications, which have obviously not been developed with support for run-time repartitioning in mind, must partially be rewritten such that they become compatible with the programming model of the new middleware platform. Instead, our new approach is to develop a byte code transformer that transparently injects new functionality to an existing distributed Java application such that this application becomes automatically run-time repartitionable by the monitor. The motivation behind this approach taken is that programmers do not want to distort their applications to match the programming model of whatever new middleware platform.

Finally, an important benefit of our mechanism for capturing distributed threads is its **portability**: (1) byte code transformations integrate the required functionality transparently into existing Java applications. A custom class loader can automatically perform the byte code transformation at load-time. (2) Our serialization mechanism does require no modifications of the JVM. This makes the implementation portable on any system, as long as a standard JVM is installed on that system. (3) Our mechanism does require no modifications of the underlying ORB. Our mechanism works seamless on top of any ORB with an RPC–like programming model, provided that our byte code transformation is performed before stub code generation.

However, a limitation is that our serialization mechanism is only applicable on top of a dedicated cluster of machines where network latencies are low and faults are rare. This is not directly a dependability of our approach, but rather a dependability of the RPC-like programming model: performing blocking calls on remote objects is after all only feasible on a reliable, high-bandwidth and secure network. As such our serialization mechanism in not well suited for Internet or wireless applications.

1.2. Structure of the paper. This paper is structured as follows: first, in section 2 we discuss the problems with using Brakes for capturing distributed threads. In section 3 we introduce the notion of distributed thread identity as a concept to handle a distributed control flow as one computational entity. In section 4 we apply the concept of distributed thread identity to easier reuse Brakes for serialization of distributed threads. We further explain how serialization of execution-state can be initiated by an external control instance. In section 5 we introduce our prototype for run-time repartitioning and demonstrate how it works by means of a concrete example application. In section 6 we evaluate the performance overhead and byte code blowup that is generated by our approach. Section 7 discusses related work. Finally we conclude and look to future work in section 8.

2. Problem statement. In this section we first shortly describe the implementation of Brakes, our earlier developed mechanism for serialization of JVM threads. Then we discuss the problem we encountered when trying to reuse Brakes for capturing distributed execution-state.

2.1. Brakes for JVM serialization. In Brakes the execution-state of a JVM thread is extracted from the application code that is executing in that thread. For this, a byte code transformer inserts capture and reestablishing code blocks at specific positions in the application code. We will refer to this transformer as the Brakes transformer.

With each thread two flags (called isSwitching and isRestoring) are associated that represent the execution mode of that specific thread. When the isSwitching flag is on, the thread is in the process of capturing its state. Likewise, a thread is in the process of reestablishing its state when its isRestoring flag is on. When both flags are off, the thread is in normal execution. Each thread is associated with a separate Context object into which its execution-state is switched during capturing, and from which its execution-state is restored during reestablishing.

The process of capturing a thread's state (indicated by the empty-headed arrows in Fig. 2.1) is then implemented by tracking back the control flow, i.e., the sequence of nested method invocations that are on the stack of that thread. For this the byte code transformer inserts after every method invocation a code block that switches the stack frame of the current method into the context and returns control to the previous method on the stack, etc. This code block is only executed when the isSwitching flag is set. The process of reestablishing a

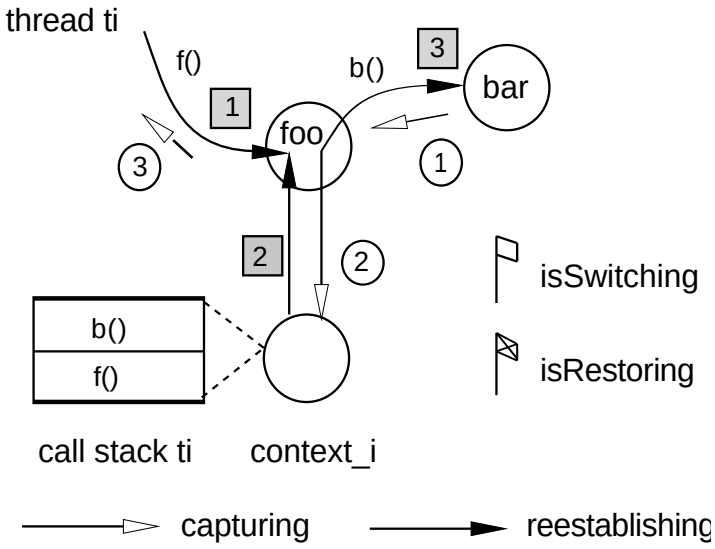


FIG. 2.1. *Thread Capturing/Reestablishing in Brakes.*

thread's state (indicated by the full-headed arrows in Fig. 2.1) is similar but restores the stack frames in reverse order on the stack. For this, the byte code transformer inserts in the beginning of each method definition a code block that restores stack frame data of the current method and subsequently creates a new stack frame for the next method that was on the stack, etc. This code block is only executed when the isRestoring flag is set.

A context manager per JVM manages both Context objects and flags. The inserted byte codes switch/restore the state of the current thread into/from its context via a context-manager-defined static interface. The context manager manages context objects on a per JVM thread basis. So every thread has its own Context object, exclusively used for switching the state of that thread. The context manager looks up the right context object with the thread identity as hashing key. For more information about Brakes, we refer the reader to [15].

2.2. Problem with Brakes to capture distributed execution-state. This section describes the problem that we encountered when trying to reuse Brakes for capturing distributed execution-state. In Brakes, execution-state is saved per local JVM thread. This works well for capturing local control flow but not for capturing a control flow that crosses system boundaries. Fig. 2.2 illustrates the problem. Once thread ti in

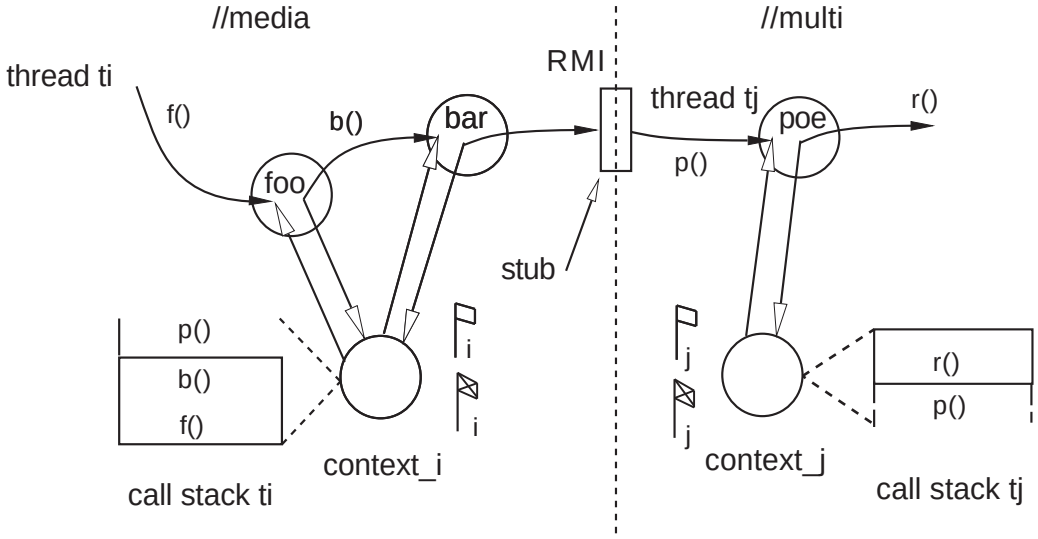


FIG. 2.2. *Context per JVM Thread.*

the example reaches method `b()` on object bar, the call `p()` on object poe is performed as a remote method invocation. This remote call implicitly starts a new thread tj at host media. Physically, the threads ti and tj hold their own local subset of stack frames, but logically the total set of frames belongs to the same distributed control flow. The context manager however, is not aware of this logical connection between threads ti and tj. As a consequence Brakes will manage contexts and flags of these JVM threads as separate computational entities, although they should be logically connected. Without this logical connection, it becomes difficult to robustly capture and reestablish a distributed control flow as a whole entity. For example, it becomes quasi impossible for the context manager to determine the correct sequence of contexts that must be restored for reestablishment of a specific distributed control flow.

3. Distributed threads and distributed thread identity. The above discussed problem is a specific instance of a more general problem. The essence of this problem is that we can not rely on JVM thread identity to refer to a distributed thread as one and the same computational entity. We now discuss how we have extended Java programming to cope with this problem.

A Java program is executed by means of a Java Virtual Machine thread (JVM thread). Such a thread is the unit of computation. It is a sequential flow of control within a single address space (i.e. JVM). However, distributed threads [4] execute as flows of control that may cross physical node boundaries. A distributed thread is a logical sequential flow of control that may span several address spaces (i.e. JVMs). As shown in Fig. 3.1 a distributed thread T is physically implemented as a concatenation of local (per JVM) threads [t1,...,t4] sequentially performing remote method invocations when they transit JVM boundaries. In the context of

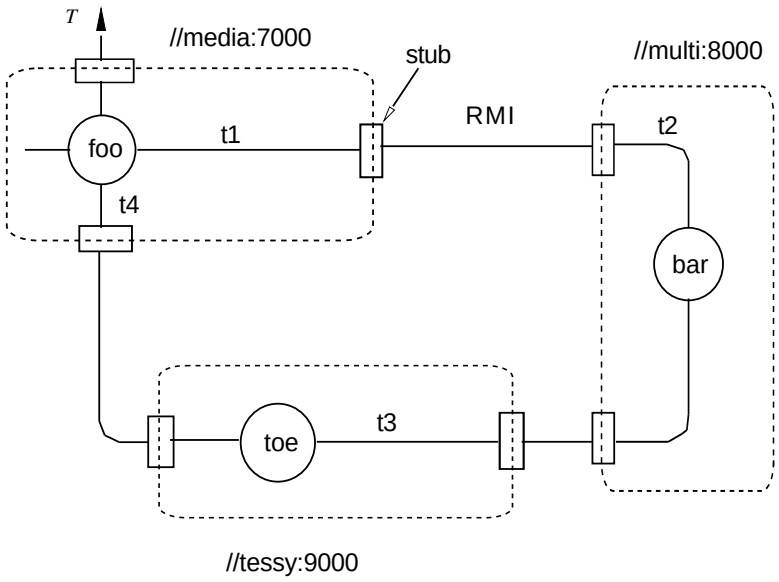


FIG. 3.1. *A Distributed Thread.*

a distributed control flow programming model, distributed threads offers a general concept for a distributed computational entity.

In a local execution environment, i.e., for centralized programs that run on one JVM, the JVM thread identifier offers a unique reference for a single computation entity. For a distributed application however distributed threads execute as flows of control that may cross physical node boundaries. Once the control flow crosses system boundaries a new JVM thread is used for continuing execution and so logical thread identity is lost. As a consequence JVM thread identifiers are too fine grained for identification of distributed threads. We extend Java programs with the notion of distributed thread identity. Propagation of a globally unique distributed thread identity provides a uniform mechanism by which all the program's constituent objects involved in a distributed thread can uniquely refer to that distributed thread as one and the same computational entity.

With respect to the subject of this paper, the implementation of distributed threads should enforce the following two rules:

1. A distributed thread identity must be created once, more specifically when the distributed thread is scheduled for the first time.

2. A distributed thread identity must not be modified after it is created. This is because it must provide physically dispersed objects with a unique and immutable reference to a distributed thread.

We used M. Dahm's tool for byte code transformation, BCEL [5], to develop a byte code transformer that extends Java programs with the notion of distributed threads, more specifically distributed thread identity. Hereafter we will refer to this transformer as the DTI transformer.

To achieve propagation of distributed thread identities, the DTI transformer extends the signature of each method with an additional argument of class D_Thread_ID. D_Thread_ID is a serializable class that implements an immutable, globally unique identifier. The signature of every method invoked in the body of the methods must be extended with the same D_Thread_ID argument type too. For example, a method `f()` of a class C is rewritten as:

```
//original method code      //transformed method code
f(int i, Bar bar) {         f(int i, Bar bar, D_Thread_ID id) {
 ...                         ...
 bar.b(i);                   bar.b(i, id);
 ...                         ...
}                           }
```

When `f()` is called, the D_Thread_ID is passed as an actual parameter to `f()`. Inside the body of `f()`,

`b()` is invoked on bar. On its turn, the body of `f()` passes the D_Thread_ID it received as an extra argument to `b()`. This way the distributed thread identity is automatically propagated with the control flow along the method call graph.

3.1. Creation and modification of a distributed thread. The Java thread programming model offers the application programmer the possibility to start up a new thread from within the `run()` method of an object of a class that implements the java.lang.Runnable interface. The DTI transformer instrument this kind of classes such that they implement the D_Runnable interface instead. D_Runnable is defined as follows:

```
interface D_Runnable {
 void run(D_Thread_ID id);
}
```

The example class Bar that originally implements the Runnable interface illustrates this transformation:

```
//original class definition    //transformed class definition
class Bar implements           class Bar implements
 java.lang.Runnable {           D_Runnable {
 ...                            ...
 void run() {...}               void run(D_Thread_ID id) {...}
}                              }
```

As stated in section 3, the identity of a distributed thread must be created at the moment the distributed thread is created. This behavior is encapsulated in the D_Thread class, which wraps each D_Runnable object in a D_Thread and as such serves as an abstraction for creating a new distributed thread. A new distributed thread can simply be started with:

```
Bar b = new Bar();
D_Thread dt = new D_Thread(b);
dt.start();
```

The D_Thread class itself is defined as:

```
class D_Thread implements java.lang.Runnable {
  public static D_Thread_ID getCurrentThreadID() {
  }
  public D_Thread(D_Runnable o) {
    object = o;
    id = new D_Thread_ID();
  }
  public void run() {
    object.run(id);
  }
  public void start() {
    new Thread(this).start();
  }
  private D_Runnable object;
  private D_Thread_ID id;
}
```

As stated in section 3, distributed thread identities may never be modified. As such D_Thread does not provide any method operations for this. Furthermore, D_Thread_ID objects are stored either as a private instance member of class D_Thread or as a local variable on a JVM stack. Nonetheless it remains possible for

a malicious person to modify distributed thread identities, e.g., by inserting malicious byte code that modifies the value of the local variable pointing to a D_Thread_ID. To prevent this, additional measures are necessary, which are subject of future work.

3.2. Inspection of distributed thread identity. Since distributed thread identities are propagated with method invocations as an additional last argument, it is possible to compute for every method definition which local variable on the JVM stack points to the corresponding D_Thread_ID object. This allows a management subsystem to inspect the value of the local D_Thread_ID variable at any point in the method code. This inspection requires two parts. First the management subsystem must define a static interface that includes the D_Thread_ID as an argument of its methods. Second the application code must be extended with byte codes at specific marking points in its program that invoke these methods. Then, each time a distributed thread passes such marking point, it notifies the management subsystem, identifying itself by means of the passed D_Thread_ID argument.

Our implementation of distributed thread identity also allows application objects to inspect D_Thread_ID, by means of a static operation `getCurrentThreadID()`. The implementation of this method is encapsulated in the DTI transformer, which transparently replaces any invocation of this method with the value of the D_Thread_ID variable.

3.3. Integration with the ORB architecture. Applying the byte code transformation to applications developed with an off-the-shelf ORB demands some attention. The programmer must be aware of generating the stub classes for the different remote interfaces only after byte code transformation has been applied. This to make sure that stubs and skeletons would propagate distributed thread identity appropriately.

4. Distributed thread serialization. Distributed thread identity provides an elegant solution to the problem of serializing distributed threads. More specifically if we augment the Brakes transformer with functionality for distributed thread identity inspection (i.e. D_Thread_ID) by the context manager via a static interface (see section 2.1), it is possible to implement a management subsystem that manages contexts on a per distributed thread basis (see Fig. 4.1). This update to the Brakes transformer entails only slight modification.

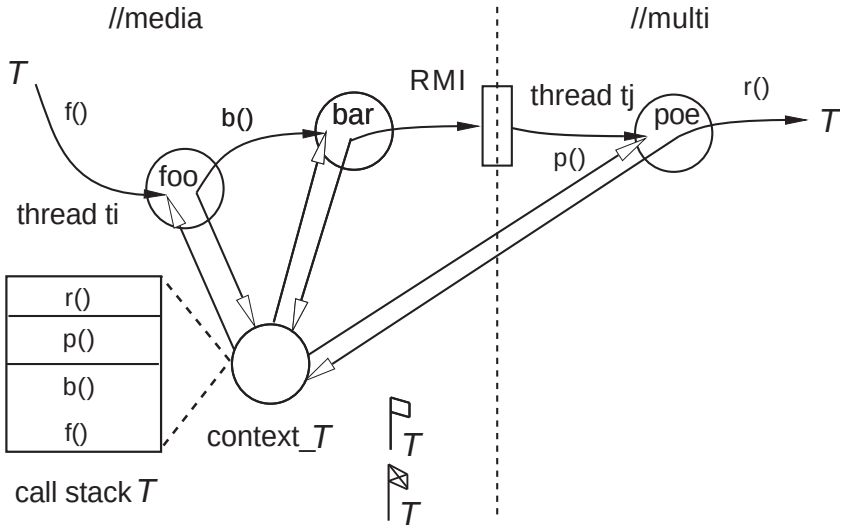


FIG. 4.1. *Context per Distributed Thread.*

We now discuss these extensions.

4.1. New static interface for inspecting distributed thread identity. The Brakes byte code transformer must be modified to support inspection of distributed thread identity by the management subsystem. As stated in the description of Brakes, JVM thread stack frames are switched into the associated Context object via a static interface defined by the context manager. For example, switching an integer from the stack into the context is done by inserting in the method's byte code at the appropriate code position an invocation of the following static method:

```
static curPushInt(int i) {
 Context c = getContext(Thread.currentThread());
 c.pushInt(i);
}
```

Such push-methods are defined for all basic types as well as for object references. Complementary, there are pop-methods for restoring execution-state from the context. The appropriate target Context object is looked up with the current thread identity as hashing key. However, in order to allow the context manager to manage Context objects on a per distributed thread basis, this static interface must be changed such that the context manager can inspect the identity of the current distributed thread. Thus:

```
public static pushInt(int i, D_Thread_ID id) {
 Context c = getContext(id);
 c.pushInt(i);
}
```

Note that byte code instructions must be inserted for retrieving the D_Thread_ID local variable before calling the static methods.

4.2. Efficient management of serialized distributed execution-state. Extending the static interface of the Brakes context manager allow us to build an associated distributed management subsystem, illustrated in Fig. 4.2, that manages captured execution-state on a per distributed thread basis. The management subsystem

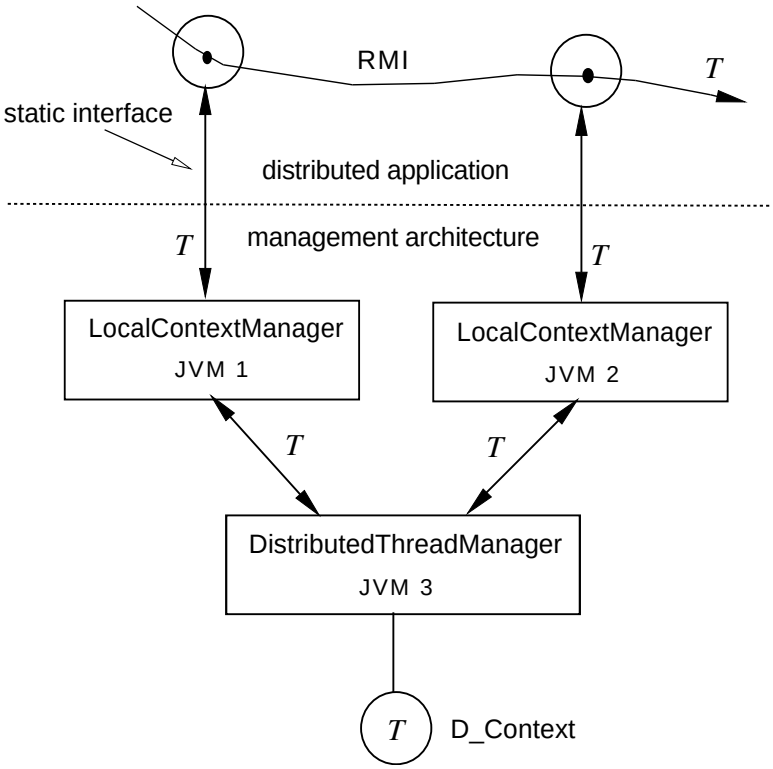


FIG. 4.2. *Distributed Architecture of the Context Manager.*

consists of a context manager for each JVM where the distributed application executes, to which we will refer as *local context manager*. Capturing and restoring code blocks still communicate with the static interface of the local context manager, but the captured execution-state is now managed per distributed thread by one central manager, *the distributed thread manager*.

To further deal with the problem that Brakes is not designed for capturing distributed execution-state, we also had to rearrange the management of the isSwitching and isRestoring flags. First, while in Brakes there was a separate isSwitching and isRestoring flag for each JVM thread, we now manage only one isSwitching and one isRestoring flag for the entire distributed execution-state of the application. Both flags are stored as static global variables on the distributed thread manager and are replicated with strong consistency on each local context manager. Furthermore we introduced a new flag, isRunning, associated with each individual distributed thread that marks the start of capturing and the end of reestablishing the execution-state of that distributed thread.

A positive side effect of the rearrangement of flags is that we drastically reduce the overhead during normal execution. When inspecting the global isSwitching and isRestoring flags during normal execution, no costly hashtable look up on distributed thread identity is performed anymore. Only during capturing and reestablisment the isRunning flag is looked up with D_Thread_ID as hashing key; however these look ups do not occur during normal execution. This choice may seem to be a trade–off between efficiency and flexibility. The rearrangement of flags results in a less flexible mechanism that can only capture execution-state at the level of the whole distributed execution state of the application. It is not possible to capture one distributed thread, without stopping other distributed threads. However, this coarse-grained scale is exactly what we want: it does not make sense to capture one thread, without stopping another thread when they are executing in the same application objects. In section 6.2 we discuss performance overhead more in detail.

4.3. External initiation of capturing and reestablishing. The Brakes JVM thread serialization mechanism is designed in the context of mobile agents and as such it is not designed for being initiated by an external control instance. To deal with this problem, we extended the Brakes transformer to insert extra byte codes at the beginning of each method body to verify whether there has been an external request for capturing the execution-state. We will refer to this code as {external capturing request check}. Furthermore, the distributed thread manager offers a public interface that enables an external control instance to initiate the capturing and reestablishing of distributed execution-state. Capturing of execution-state is started by calling the operation `captureState()` on the distributed thread manager. This method sets the isSwitching flag on all local context managers through broadcast. As soon as a distributed thread detects the isSwitching flag is set, (inside the first executed {external capturing request check} code) the distributed thread sets off its isRunning flag and starts switching itself into its context. Reestablishment of execution is initiated by calling the operation `resumeApplication()` on the distributed thread manager. This method sets the isRestoring flag on each local context manager and restarts the execution of all distributed threads. Each distributed thread detects immediately that the isRestoring flag is set, and thus restores itself from the context. Once the execution-state is reestablished the distributed thread sets on its isRunning flag (inside the {external capturing request check}) and resumes execution. When all distributed threads execute again, the distributed thread manager turns off the isRestoring flag on all local context managers through broadcast.

5. Run–time repartitioning at work. In this section we present our prototype for run-time repartitioning and demonstrate it for a simple text translator application. First we describe the process of run-time repartitioning. Then we give a sketch of the prototype. Next we illustrate the byte code transformations. Finally we explain the process of run-time repartitioning by means of an example.

5.1. The four phases of run-time repartitioning. run-time repartitioning aims to improve the global load balance or network communication overhead by repartitioning the object configuration of the application over the available physical nodes at run-time. We distinguish between 4 successive phases in the run-time repartitioning process. In the first phase, a load balancing monitor allows an administrator to monitor the application's execution and let him decide when to relocate the application objects over the available physical nodes. In the second phase, the monitor invokes the distributed thread manager to capture the distributed execution-state of the application. After this, the execution of all application objects is temporarily suspended and the corresponding distributed thread states are stored as serialized data in the context repository of the distributed thread manager. In the third phase, the monitor carries out the initial request for repartitioning by migrating the necessary objects over the network. In the final and fourth phase, the monitor invokes the distributed thread manager to reestablish the distributed execution-state of the application. As soon as the execution-state is reestablished, the application continues where it left off.

5.2. Prototype. In the run-time repartitioning prototype, the serialization mechanism is integrated with a simple load balancing monitor. We demonstrate the repartitioning prototype for a simple text translator

system (see Fig. 5.1). The text translator is composed with a number of objects that can be distributed over some hosts. For each translation a new distributed thread is started. A client sends the text with a source and target language to a Server object. The Server forwards the job to a ParserBuilder, who sends each sentence for translation to a Translator. The Translator uses a Dictionary object for the translation of individual words. As soon as a sentence is translated the Translator returns it to the ParserBuilder. The ParserBuilder assembles the translated text. Finally the translated text is returned to the Server who sends it back to the client. Fig. 5.2 gives

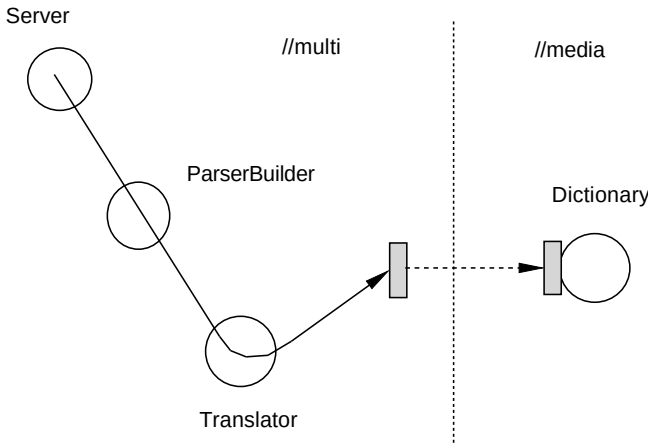


FIG. 5.1. *Prototype for Run–time Repartitioning.*

a snapshot of the load balancing monitor. The monitor offers a GUI that enables an administrator to do run-time repartitioning. The left and middle panels show the actual object distribution of the running application. The panels on the right show the captured context objects per distributed thread after a repartitioning request.

Since passive object migration, i.e., code and data migration, but no migration of run-time information like the program counter and the call stack, is necessary during the third phase of the run-time repartitioning process we used the mobile object system Voyager 2.0 [9] as distributed programming model in our prototype. To enable capturing and reestablishment of distributed execution-state during phases two and four, the im-

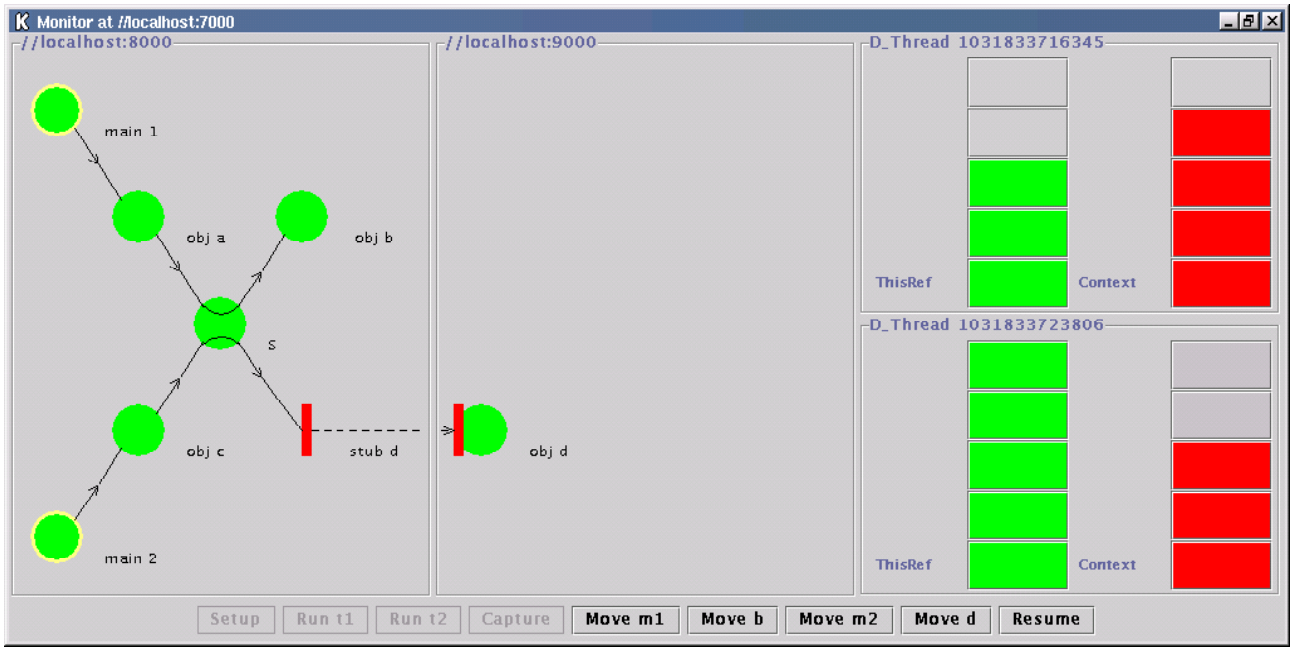


FIG. 5.2. *Snapshot of the Run–time Repartitioning Monitor.*

plementation code of the text translator system must be hauled through our byte code transformer. Finally, the distributed architecture of the associated management subsystem as shown in Fig. 4.2, defines an abstract framework that must be instantiated by a concrete distributed implementation. We used Voyager for this too, but another distribution platform like Java RMI was also possible. Thus in our prototype LocalContextManager and DistributedThreadManager are implemented as Voyager objects.

5.3. Byte code transformations. Before illustrating the process of run-time repartitioning we first give an overview of the transformation of the application code. We limit the extract to the principal code as in Fig. 5.3. The Italic marked code is inserted byte code. Each method signature as well as each method invocation is extended with an extra D_Thread_ID argument by the DTI transformer. The Brakes transformer has inserted code blocks for capturing the execution-state of a distributed thread (*switching code block*) and restoring it after-wards (*isRestoring code block*).

```
class Translator {
 Sentence analyze(Sentence sentence, D_Thread_ID, threadID) {
    {isRestoring code block}
    {external capturing request check}
    {switching code block}
    Sentence tSentence = new Sentence();
    Word word, tWord;
    sentence.resetCursor(threadID);
    {switching code block}
    while(!sentence.endOfSentence(threadID)){
      {switching code block}
      word = sentence.next(threadID);
      {switching code block}
**    tWord = dictionary.translate(word, threadID);
      {switching code block}
      tSentence.add(tWord, threadID);
      {switching code block}
    }
    return tSentence;
 }
 ...
 private Dictionary dictionary;
}

class Dictionary {
* public Word translate(Word word, D_Thread_ID threadID) {
    {isRestoring code block}
    {external capturing request check}
    {switching code block}
    ...
  }
  ...
}
```

FIG. 5.3. *Application code after Byte code Transformation.*

5.4. An example of run-time repartitioning. We now explain the process of run-time repartitioning starting from Fig. 5.1. Suppose the administrator decides to migrate the Dictionary object during the translation of a text. At a certain moment, lets say when the control flow enters the `translate()` method in Dictionary, the administrator pushes the capture button on the monitor. This point is marked in Fig. 5.3 with *. At that moment the execution-state of the distributed thread is scattered over two hosts as illustrated in Fig. 5.4. Pushing

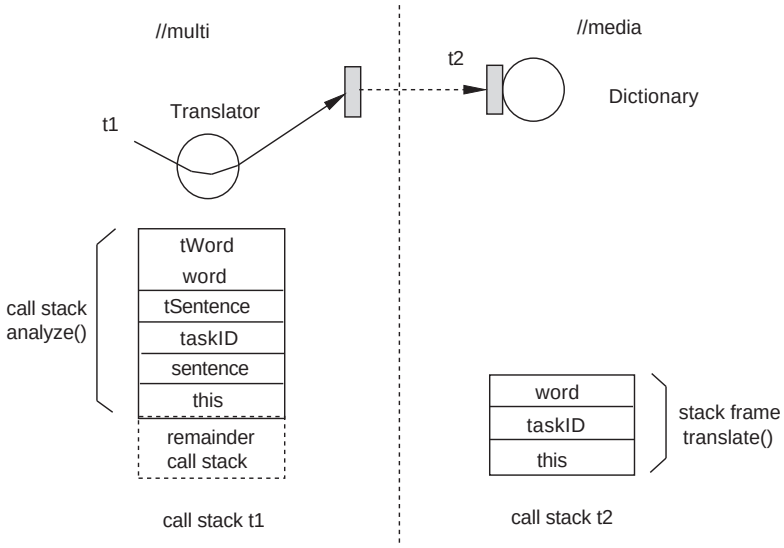


FIG. 5.4. *Distributed Execution—state before Capturing.*

the capture button invokes `captureState()` on the distributed thread manager that sets the isSwitching flag. The distributed thread detects this in the {external capturing request check} code. Immediately it set off its isRunning flag and saves the execution-state of the `translate()` method. This includes: (1) the stack frame for `translate()` (i.e. the only frame for thread t2 on the call stack, see Fig. 5.4); (2) the index[1] of the last invoked method in the body of translate (i.e. zero for {external capturing request check}); (3) the object reference to the Dictionary object (i.e. the this reference). For reasons of efficiency, the execution-state is buffered per stack frame by the local context manager until completion of switching. Then the local manager forwards the serialized data to the distributed manager who stores it into the context repository of the distributed thread with the given D_Thread_ID.

The last instruction of the {switching code block} is a return. This redirects the control to the previous frame on the call stack of the distributed thread, in our case the `analyze()` method. The control flow returns from host to host (i.e. from media to multi, see Fig. 5.4) which means that the execution-state of the JVM thread t2 now is completely saved. In the code of Fig. 5.3, we then reach the point marked as $**$. Next the execution-state for `analyze()` is saved, i.e.: (1) the stack frame for `analyze()` (i.e. the top frame of JVM thread t1 as in Fig. 5.4): (2) the index of the last invoked method in the body of analyze (i.e. 4, see Fig. 5.3); (3) the object reference to the Translator object. As soon as the buffered data is written to the context repository of the distributed thread manager, another return at the end of the {switching code block} redirects the control flow to the previous frame on the call stack. Subsequently, the execution-state for that method is saved. This process recursively continues until the JVM thread t1 returns to the `run()` method of D_Thread. At that time the DistributedContext contains the complete distributed execution-state of the distributed thread.

Once the complete distributed execution-state is saved the Dictionary object can be migrated from media to multi. To this purpose the administrator pushes the corresponding migrate-button on the monitor. As explained in section 5.2, we used Voyager as distribution platform for our prototype. Voyager dynamically transfers the object references from remote to local and vise verse.

[1] this index refers to the number the Brakes transformer associates with the subsequent invoke instructions in the body of each method, starting with 0 for external capturing request check code, 1 for the first invoke instruction and so on; the index of the last performed invoke instruction is saved in the context to remember which methods where on stack

Once the repartitioning is ready the administrator can push the resume button which invokes `resumeApplication()` on the distributed thread manager. That turns off the isSwitching flag and sets the isRestoring flag. Next a new JVM thread is created at multi to resume the translation. During the reestablishing process the relevant methods are called again in the order they have been on the stack when state capturing took place. The new thread takes the original D_Thread_ID with it. This is the key mechanism for reconstructing the original call stack. Each time inserted byte code reestablish the next stack frame the managers use the distributed thread identity to select the right context object for that particular distributed thread.

Fig.5.5 illustrates the reestablishment of the last two frames in our example. When `analyze()` on Translator is invoked, the isRestoring code block will be executed, based on the actual state of the flags (isRestoring = on, isRunning = off). The inserted byte code restores the values of the local variables of the `analyze()` frame one by one via the local context manager (see the left part of Fig. 5.5). At the end the index of the next method to invoke is picked up. For the `analyze()` frame an index 4 was earlier saved, so the fourth method, i.e. `translate()`, must be invoked on the Dictionary object. The managers pickup the reference to this object and `translate()` will be invoked. Again the isRestoring code restores the local execution-state of the `translate()` method (the right part of Fig. 5.5). This time the index for the next method is zero. This is the signal for the context manager to reset the isRunning flag of the current distributed thread and to resume its normal execution. At this point the expensive remote interaction between the translator and the dictionary objects is transferred into a cheap local invocation. Note that in the example the execution-state is captured with the

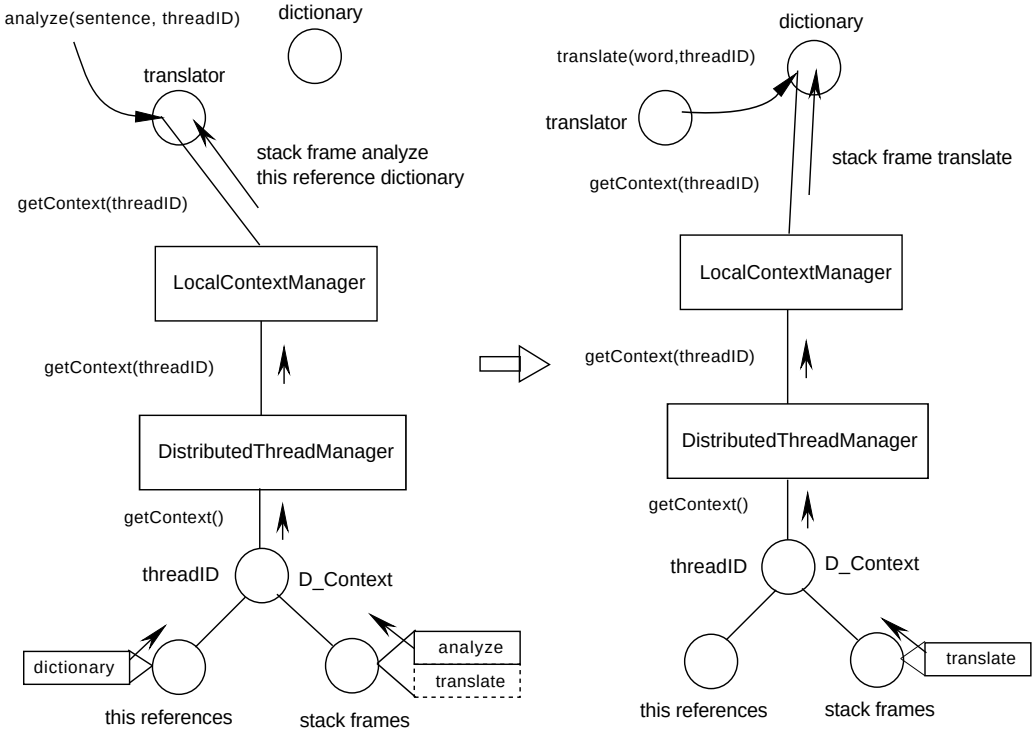


FIG. 5.5. *Reestablishing a Distributed Execution–state.*

translator object waiting on a pending reply of a remote method invocation (i.e. `dictionary.translate(word, threadID)`, see Fig. 5.3). This illustrates the advantage of phases two and four of our run-time repartitioning scheme in keeping execution and object migration completely orthogonal to each other.

6. Evaluation. In this section we evaluate the serialization mechanism for a distributed execution-state. Since inserting byte code introduces time and space overhead we look to the blowup of the class files and give results of performance measurements. To get a representative picture, we did tests on different types of applications. At the end we look at the limitations of our current implementation and outline the restrictions of our model.

6.1. Blowup of the byte code. The blowup of the byte code for a particular class highly depends on the number and kind of defined methods. Since the Brakes transformer inserts code for each invoke-instruction that occurs in the program, the space overhead is directly proportional to the total number of invoke-instructions that occur in the application code. Per invoke-instruction, the number of additional byte code instructions is a function of the number of local variables in the scope of that instruction, the number of values that are on the operand stack before executing the instruction and the number of arguments expected by the method to be invoked. The DTI transformer rewrites method and class signatures. This adds a space overhead proportional to the number of signature transformations. We measured the blowup for three kinds of applications:

1. Low degree of method invocation, i.e., the program has a structure `main{m1;}` thus the code is compacted in one method body;

2. Nested method invocations, i.e., the program has a structure `main{m1;}; m1{m2; m3;}; m3{m4;}` thus the code is scattered over a number of nested methods;

3. Sequential method invocations, i.e., the program has a structure `main{m1; m2; m3; m4;}` thus the code is scattered over a number of sequential non-nested methods.

Fig. 6.1 shows the results of our measurements. The dark part of the bars represents the size of the original code. The gray part represents the additional code for distributed thread functionality, while the white part represents the extra added code for full serialization functionality. We measured an average blowup for distributed thread identity of 27 % and 83 % for full serialization functionality. The expansion for Sequential is rather high, but its code is a severe test for blowup.

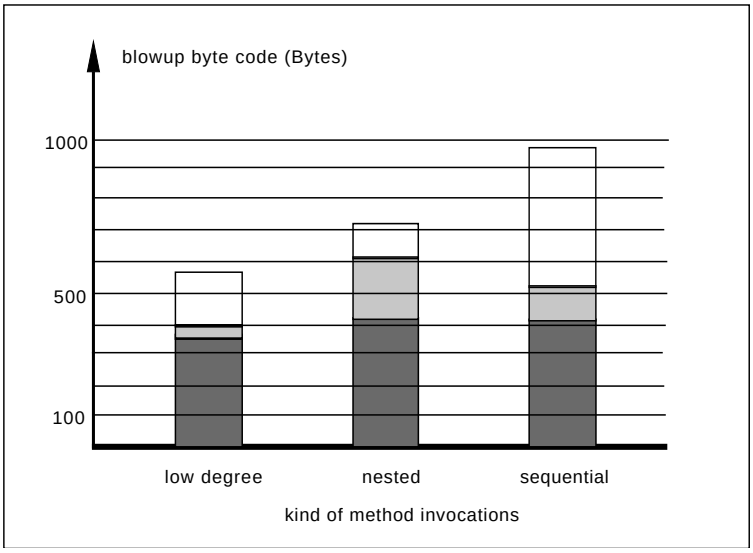


FIG. 6.1. *Byte code Blowup for Three kinds of Applications.*

6.2. Performance measurements. For performance measurements, we used a 500 MHz Pentium III machine with 128 MB RAM with Linux 2.2 and the SUN 2SDK, JIT enabled. We limited our tests to the overhead during normal execution. This overhead is a consequence of the execution of inserted byte code. Fig. 6.2 shows the results of our tests. The dark bars represent the execution speed of the original application code; the gray parts represent the overhead due to byte code transformation for distributed thread identity, while the white parts represents the additional overhead for distributed thread serialization. We measured an average overhead of 3 % for distributed thread identity. For full serialization functionality we get an average overhead of 17 %, a quite acceptable result. Note that "normal" applications typically are programmed in a nested invocation style. As such, the results for the Nested application are a good indication for blowup and performance overhead in practice. It is difficult to compare our measurement results with other systems, since to our knowledge, no related system truly covers functionality for serialization of *distributed* execution-state as our system does. In section 7 we discuss related work.

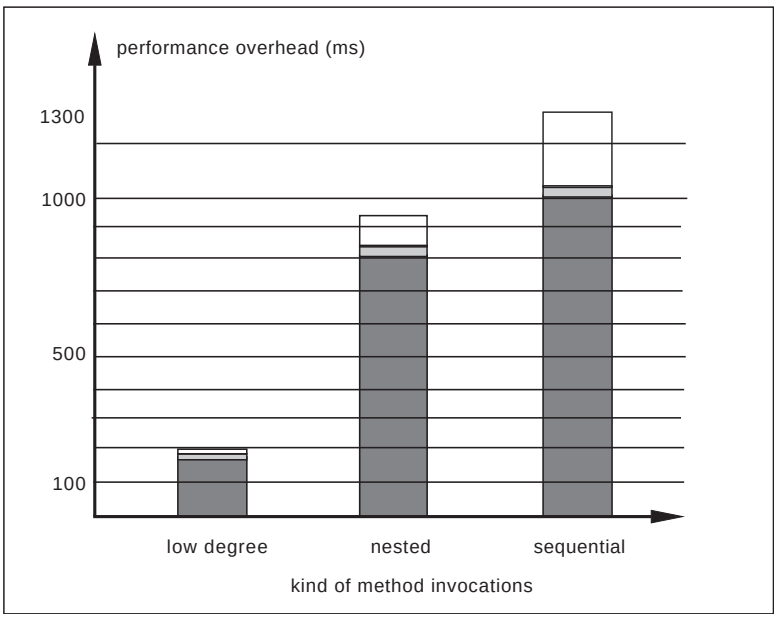


FIG. 6.2. *Performance Overhead for Three kinds of Applications.*

6.3. Limitations of the current implementation. Our byte code transformers have some limitations we intend to eliminate in the future. Although possible, we have not yet implemented state capturing during the execution of an exception handler. The major difficulty here is dealing with the `finally` statement of a `try` clause. Currently our byte code transformer throws away all debugging information associated with a Java class. This affects the ability to debug a transformed class with the source-code debugger. Furthermore, the DTI byte code transformer encapsulates each user defined JVM thread into a D_Thread, but currently ignores other JVM thread related code. Thus our current model doesn't support aspects such as thread locking, e.g. in synchronized code sections.

6.4. Restrictions of the model. Our model is intended and only applicable for applications running on top of a dedicated cluster of machines where network latencies are low and faults are rare. This is not directly a dependability of our approach, but rather a dependability of the RPC-like programming model: performing blocking calls on remote objects is after all only feasible on a reliable, high-bandwidth and secure network. Furthermore, in section 4.2 we already mention that the granularity of our repartitioning algorithm is at JVM level. As soon as the isSwitching flag is set all running distributed threads are suspended together irrespective of whatever application they belong. Thus applications that are transformed with our byte code transformer and that execute on the set of involved Java Virtual Machines will be suspended together.

Since we extract thread execution-state at byte code level we cannot handle a method call that causes a native method to be placed on the thread stack. Therefore programs that use reflection will not work properly with our repartitioning model.

In our prototype we transformed the application classes before deployment, but it is possible to defer this byte code transformation until run-time. In Java, this can easily be realized by implementing a custom class loader that automatically performs the transformation. In this regard, the overhead induced by the transformation process (which is not small for the current implementation) becomes a relevant performance factor.

7. Related work. The discussion on related work is organized according to the related fields that touches our work.

7.1. Distributed Threads. Existing work [4] in the domain of (real-time) distributed operating systems has already identified the notion of distributed thread as a powerful basis for solving distributed resource management problems. D. Jensen at CMU introduced the notion of distributed thread in the Alpha distributed real-time OS kernel. The main goal of distributed threads in the Alpha kernel was integrated end-to-end

resource management based on propagation of scheduling parameters such as priority and time constraints. In our project we adopted the notion of distributed thread (identity) at the application level. This allows the distributed management subsystem to refer to a distributed control flow as one and the same computational entity.

7.2. Strong Thread Migration. Several researchers developed mechanisms for strong thread migration in the context of Mobile Agents. Stefan Fünfrocken at TU Darmstadt [7] has implemented a transparent serialization mechanism for local JVM threads by processing the source code of the application. In this approach the Java exception mechanism is used to capture the state of an ongoing computation. A source code transformation requires the original Java files of the application. Besides, it is much easier to manipulate the control flow at byte code level than at source code level. As a result byte code transformation is much more efficient especially in terms of space overhead. Sakamoto et al. [11] developed a transparent migration algorithm for Java application by means of byte code transformation. They too used the exception mechanism of Java to organize the serialization of the execution-state. We have chosen not to use the exception mechanism, since entries on the operand stack are discarded when an exception is thrown, which means that their values cannot be captured from an exception handler. Sakamoto et al. solved this problem by copying all those values in extra local variables before method invocation, but this causes much more space penalty and performance overhead. In her dissertation [17], Wei Tao proposes another portable mechanism to support thread persistence and migration based on byte code rewriting and the Java exception mechanism. Contrary to the others, this mechanism also works for applications that use synchronization and locks.

7.3. Multi-Threading for Distributed Mobile Objects in FarGo. Abu and Ben-Shaul integrated a multi-threading model for distributed and mobile objects in the FarGo framework [1]. A FarGo application consists of a number of 'complets'. Complets are components similar to components in other frameworks, and in addition they are the unit of relocation in the model. The distributed mobile thread model of FarGo is based on a thread-partitioning scheme. Programmers must mark a migratable complet as thread-migratable (T-Migratable) by implementing the empty T_Migratable interface. The FarGo compiler uses this interface to generate proper thread partitioning code. Thread partitioning is integrated in the complet reference architecture, i.e. a stub and a chain of trackers. When a T_Migratable complet is referenced the invoking thread waits, and a new thread is started in the referenced complet. The migration itself is based on the source code transformation of Fünfrocken' migration scheme [7]. FarGo introduces a new distributed programming model to support migration, while our mechanism can be used for any existing Java RMI application.

7.4. Byte Code Transformations for Distributed Execution of Java applications. The Doorastha system [6] allows implementing fine-grained optimization's for distributed applications just by means of code annotations. Doorastha is not an extension or a super–set of the Java language but instead is based on annotations to pure Java programs. By means of these annotations it is possible to dynamically select the required semantics for distributed execution. This allows to develop a program in a centralized (multi-threading) setting first and then prepare it for distributed execution by annotation. Byte code transformation will generate a distributed program whose execution conforms to the selected annotations. Researchers at the University of Tsukuba, Japan [14] developed a system named Addistant, which also enables the distributed execution of a Java program that originally was developed to run on a single JVM. We share the common point of view with the researchers that software developers often prefer to write software apart of non-functional aspects. It's only at deploy-time that those aspects have to be integrated, preferable in a transparent way. To avoid deadlock in the case of call back, Addistant guaranties that local synchronized method calls of one distributed control flow are always executed by one and the same JVM thread. Therefore it establishes a one-to-one communication channel for the threads that take part in such an invocation pattern. Such a communication channel is stored as thread local variable. In this approach it isn't necessary to pass distributed thread identity along the call graph of the distributed control flow. On the other hand for a run-time repartitioning system, thread identity must be propagated with every remote invocation anyway.

8. Conclusion and future work. In this paper we presented a mechanism for serialization of a distributed execution-state of a Java application that is developed by means of a distributed control-flow programming model such as Java RMI. This mechanism can serve many purposes such as migrating execution-state over the network or storing it on disk. An important benefit of our mechanism is its portability. It can be integrated into existing applications and requires no modifications of the JVM or the underlying ORB. However, because

of its dependability on the control-flow programming model, our mechanism is only applicable for distributed applications that execute on low latency networks where faults are rare. Our contribution consists of two parts. First we integrated Brakes, our existing serialization mechanism for JVM threads, in a broader byte code translation scheme to serialize the execution-state of a distributed control flow. Second we integrated a mechanism to initiate the serialization of the distributed execution-state from outside the application. We applied the serialization mechanism in a prototype for run-time repartitioning of distributed Java applications. Our repartitioning mechanism enables an administrator to relocate application objects at any point in an ongoing distributed computation.

Since for Java RMI-liked applications, logical thread identity is lost when the control flows crosses JVM boundaries, we introduced the notion of distributed thread identity. Extending Java programming with distributed thread identity provides a uniform mechanism to refer to a distributed control flow as one and the same computational entity.

Often byte code transformation is criticized for blowup of the code and performance overhead due to the execution of inserted byte code. Based on a number of quantitative analyses we may conclude that the costs associated with our byte code translation algorithm are acceptable. Some limitations of our serialization mechanism for a distributed execution-state have to be solved. Finally it 's our intention to build a complete tool for run-time repartitioning for distributed Java RMI applications. Therefore we have to extend our current monitoring and management subsystem with several other features such as functionality for dynamic adaptation of object references after migration, support for different load balancing algorithms and an application adaptable monitor.

The latest implementation of the run-time repartitioning tool is available at:
`http://www.cs.kuleuven.ac.be/~danny/DistributedBrakes.html`

Acknowledgments. This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in industry (IWT). We would like to thank Tim Coninx and Bart Vanhaute for their valuable contribution to this work. A word of appreciation also goes to Tom Holvoet and Frank Piessens for their usefully comments to improve this paper.

REFERENCES

[1] M. Abu, I. BenShaul, *A Multi-Threading model for Distributed Mobile Objects and its Realization in FarGo*, in Proceedings of ICDCS 2001, The 21st International Conference on Distributed Computing Systems, April 16–19, 2001, Mesa, AZ, pp. 313–321.

[2] M. Baker, *Cluster Computing White Paper,* `http://www.dcs.port.ac.uk/ mab/tfcc/WhitePaper/` (2000).

[3] S. Bouchenak, *Pickling threads state in the Java system*, ERSADS 1999, Third European Research Seminar on Advances in Distributed Systems, April 1999, Madeira Island, Portugal.

[4] R. Clark, D.E. Jensen, and F.D Reynolds, *An Architectural Overview of the Alpha Real-time Distributed Kernel*, In Proceedings of the USENIX Workshop, In 1993 Winter USENIX Conf., April 1993, pp. 127–146.

[5] M. Dahm, *Byte Code Engineering*, in Proceedings of JIT'99, Clemens Cap ed., Java-Informations-Tage 1999, September 20 - 21, 1999, Düsseldorf, Germany, pp. 267–277.

[6] M. Dahm, *The Doorastha system*, Technical Report B-I-2000, Freie Universität Berlin, Germany (2001).

[7] S. Funfrocken, *Transparent Migration of Java-based Mobile Agents*, in Proceedings of the 2e International Workshop on Mobile Agents 1998, Lecture Notes in Computer Science, No. 1477, Springer-Verlag, Stuttgart, Germany, September 1998, pp. 26–37.

[8] K.C. Nwosu, *On Complex Object Distribution Technique for Distributed Computing Systems*, in Proceedings of the 6th International Conference on Computing and Information (ICCI'94), Peterborough, Ontario, Canada, May 1994, (CD-ROM).

[9] ObjectSpace Inc., *VOYAGER, Core Technology 2.0*, `http://www.objectspace.com/products/voyager/` (1998).

[10] B. Robben, *Language Technology and Metalevel Architectures for Distributed Objects*, Ph.D thesis, Deartement of Computer Science, K.U.Leuven, Belgium, ISBN 90-5682-194-6, 1999.

[11] T. Sakamoto, T. Sekiguchi, A. Yonezawa, *Bytecode Transformation for Portable Thread Migration in Java*, in Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA), Lecture Notes in Computer Science 1882, Springer-Verlag, ETH Zürich, Switzerland, September 13-15, 2000, pages 16–28.

[12] W. Shu and M. Wu, *An Incremental Parrallel Scheduling Approach to Solving Dynamic and Irregular Problems*, In Proceedings of the 24th International Conference on Parrallel Processing, Oconomowoc, WI, 1995, pages II:143–150.

[13] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley and ACM Press, ISBN 0-201-17888-5, 1998.

[14] M. Tatsubori, T. Sasaki, S. Chiba and K. Itano, *A Bytecode Translator for Distributed Execution of Legacy Java Software*, in Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001), Lecture Notes in Computer Science 2072, Springer-Verlag, Budapest, Hungary, June 18-22, 2001, pp.236–255.

[15] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen and P. Verbaeten, *Portable Support for Transparent Thread Migration in Java*, in Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents

(ASA/MA), Lecture Notes in Computer Science 1882, Springer-Verlag, ETH Zürich, Switzerland, September 13-15, 2000, pages 29–43.

[16] E. Truyen, B. Vanhaute, B. Robben, F. Matthijs, E. Van Hoeymissen, W. Joosen and P. Verbaeten, *Supporting Object Mobility, from thread migration to dynamic load balancing*, OOPSLA'99 Demonstration, November '99, Denver, USA, 1999.

[17] W. Tao, *A Portable Mechanism for Thread Persistence and Migration*, PhD thesis, University of Utah, `http://www.cs.utah.edu/ tao/research/index.html` 2001.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: September 24, 2002
Accepted: December 15, 2002

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 99–109. http://www.scpe.org
ISSN 1895-1767

DISTRIBUTED DATA MINING

VALÉRIE FIOLET[†‡], AND BERNARD TOURSEL[†],

Abstract. Knowledge discovery in databases, also called Data Mining, is an increasing valuable engineering tool. The huge amount of data to process is more and more significant and requires parallel processing.

Special interest is given to the search for association rules, and a distributed approach to the problem is considered. Such an approach requires that data be distributed to process the various parts independently. The research for association rules is generally based on a global criterion on the entire dataset. Existing algorithms employ a large number of communication actions which is unsuited to a distributed approach on a network of workstations (NOW).

Therefore, heuristic approaches are sought for distributing the database in a coherent way so as to minimize the number of rules lost in the distributed computation.

Key words. Data Mining, Association Rules, Distributed Processing, Heuristic.

1. Introduction.

Data Mining stands for the process of knowledge discovery in databases. Here, particular attention is given to the problem of association rules which exhibit a dependence on attributes of the database.

The computation of association rules has an exponential complexity with regard to the number of attributes of the database, so having recourse to parallelism could increase the size of manageable databases.

Parallel Algorithms for Shared Memory machines have been used in the past exploiting Synchronous Communication at each step of the process. However, existing parallel algorithms are clearly suited to parallel shared memory computers (SMP) because of the many synchronized communications they produce between each step of the process.

An algorithm running on a network of workstations, in which little asynchronous communication actions can be tolerated, is proposed here. The main problem of such an algorithm is now to distribute data to treat the obtained data fragments independently.

1.1. Association Rules Problem.

A well-known application of the search for association rules is the "Market-Basket Analysis" problem. The problem is to identify relationships between products that tend to be bought together.

The principal interest of the method is the clarity of the produced results, which can be easily understood by professionals of the data domain.

1.1.1. Problem definition.

Given a set of n records (each composed of items—or attributes) and m distinct items ($\in I$); the search for association rules consists in producing dependence rules to find relations between those items which predict the occurrence of other items.

An association rule is then an implication of form $X \Rightarrow Y$, where X and Y are itemsets belonging to I and where $X \cap Y = \emptyset$ [1]. (X is the condition or antecedent; Y is the conclusion or consequence).

We call k-itemset a set of k items.

The number of possible association rules is $O(m2^m)$. The computation complexity is $O(nm2^m)$.

In order to reduce the complexity of the problem, statistical measures are generally associated to computations, and help to reduce the size of the search space:

- Associated with each itemset is a **support**. The support of an itemset is the percentage of records in a database, D, which contains this itemset. (The support measures how interesting the itemset is, that is, its frequency in the dataset).
- Each association rule has an associated **confidence** measure:

$$\text{confidence}(X \Rightarrow Y) = \text{support}(X \cup Y)/\text{support}(X).$$

(The confidence of a rule measures a real causality between the condition and the conclusion).

[†]Laboratoire d'Informatique Fondamentale de Lille (UPRESA CNRS 8022), University of Lille1, Cité Scientifique, 59655 Villeneuve D'ascq CEDEX, FRANCE (tél. 03.20.43.45.39). {Fiolet, Toursel}@lifl.fr

[‡]Service Informatique—University of Mons-Hainault 6, Avenue du Champs de Mars, 7000 MONS, BELGIUM (tél. 065.37.34.46, Valerie.Fiolet@umh.ac.be)

[1]Formulation of the problem proposed by Agrawal and al. [1] and [2]

TABLE 1.1
Apriori Algorihm

Input: the database, the support threshold (minsup)
Output: F_k: set of frequent k-itemsets
(1) F_1 = frequent 1-itemsets
(2) **for**$(k = 2; F_{k-1} \neq \emptyset; k++)$**do**
(3) C_k = AprioriGen(F_{k-1})
(4) **for** each element c in C_k **do**
(5) count support for c.
(6) **end for**
(7) $F_k = \{c \in C_k | c.support \geq minsup\}$
(8) **end for**
(9) **return** $\cup F_k$

The search for association rules consists in finding rules of support and confidence greater than the given thresholds. The computation of itemsets with supports greater than the threshold is the most expensive part of a process.

A well-known algorithm for the computation of frequent itemsets is the Apriori algorithm (see Table 1.1) proposed by Agrawal and Srikant ([2], [9]). It is used as follows:

- to compute the supports of items, and then to identify frequent items (frequent 1-itemsets)
- to generate candidate 2-itemsets, to count their supports, and then to identify frequent 2-itemsets
- to generate candidate 3-itemsets, to count their supports, and then to identify frequent 3-itemsets, and so on...

The guiding principle is that: **Every subset of a frequent itemset has to be frequent**.

This principle is used to prune many candidates using the AprioriGen algorithm, which provides candidate k-itemsets (C_k) by computation on F_{k-1} (frequent (k-1)-itemsets).

1.2. Existing Parallel Algorithms. Existing parallel algorithms based on Apriori can be classified as:

- those that propose a replication of candidate k-itemsets: making it necessary to synchronize after each k^{th} step so as to communicate local supports counts. In this way, it is possible to identify globally frequent k-itemsets which are useful for the next $(k+1)^{th}$ step (Count Distribution [3][10], Parallel PARTITION [8], PDM [7]);
- those that propose a partitioning of candidate k-itemsets: making it necessary to communicate data fragments to each processor to count supports and to synchronize the process at the end of each step (Data Distribution [3][10], Intelligent Data Distribution [5], DMA [4]).

In these two kinds of parallel algorithms, it is necessary to have many synchronous communications between two steps (the k^{th} and the $((k+1)^{th})$). These kinds of solutions are clearly not suited to a system with distributed memory and costly communications.

1.3. A Distributed Solution. Computation of association rules is very expensive (the cost is exponential). Parallelism may offer a way to bring this cost under control and so permit the processing of larger databases. In two ways:

- storing the database,
- computing frequent k-itemsets on the database.

Since existing parallel algorithms use many synchronous communications, they are inappropriate on a network of workstations.

A real distributed method is proposed to solve the problem of the search for frequent itemsets and the computation of association rules.

The count of supports for itemsets could be distributed (as it is in existing parallel algorithms), but the major difficulty for a real distribution of the process is the necessity to access to the whole database to count an itemset support. Furthermore, all the information in a step (computation of 1-itemsets for example) is useful for the next step (computation of 2-itemsets, for example).

These requirements appear as a global criterion that must be taken into account. Potentially all items can appear together in an itemset; each record can have an influence on the support of an itemset.

2. An Entirely Distributed Process.

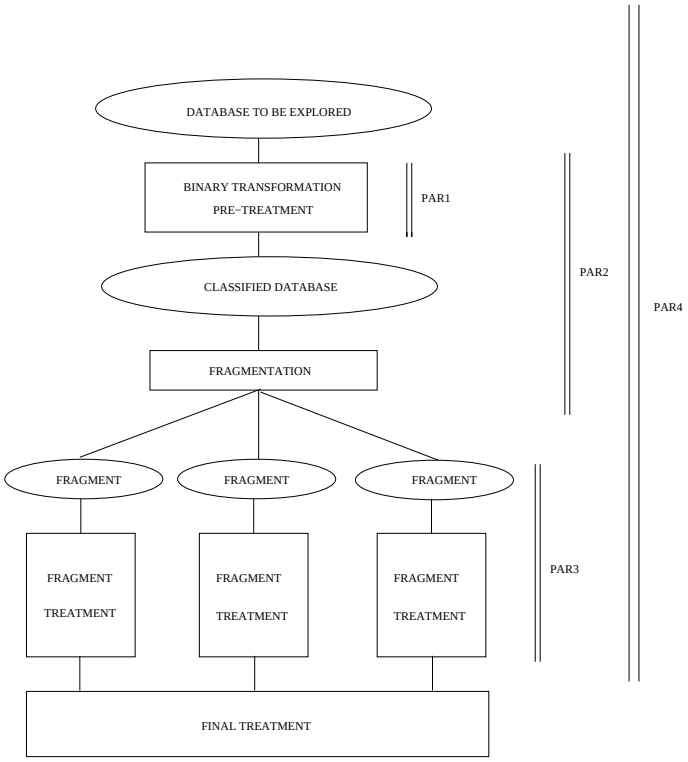


FIG. 2.1. *General schema*

2.1. General Schema. A general schema is suggested (see Figure 2.1) to find association rules on a dataset. Taking into account that a data mining process (the search for association rules, for example) is part of a KDD process (Knowledge Discovery in Data: cleaning, pre-processing, data mining, results validation ...), and that the algorithm has been tested on a real medical database, it was decided to introduce into the schema the pre-processing phase that aims to format data for the problem. Assuming that the database is initially distributed in a vertical split (non-overlapping subsets of attributes on each workstation):

- The first stage consists in pre-treating the data (see PAR1 on Figure 2.1).
- The second stage (included in PAR2 on Figure 2.1), consists in fragmenting the data to search for association rules. The quality of the distribution will influence the quality of the final results.
- The third stage (see PAR3 on Figure 2.1), computing frequent itemsets and association rules, could then run totally independently on each fragment of data, using a classical sequential algorithm for association rules (APriori, for example) on each component.
- After the end of this third step, a decision must be taken for obtaining the results (FINAL TREATMENT on Figure 2.1):
  1. either the results are considered on each fragment independently (what is linked to the profiles concept, see Sections 2.2.2 and 3);
  2. or the results are brought together and a "corrective" method is applied to them.

To distribute the processing phase (the data mining process), it was decided to execute all the processes in a distributed environment, therefore, the global PAR4 process (see Figure 2.1), consisting of computing all phases on a NOW, is considered. A component approach of the PAR4 process is adopted (see Figure 2.2). The process is broken up in three stages, each being carried out by a distinct kind of component.

- The **Pre-processing step** (the binary transformation of the database) is carried out by Clustering Objects according to the algorithm used for the binary transformation (see 2.2).

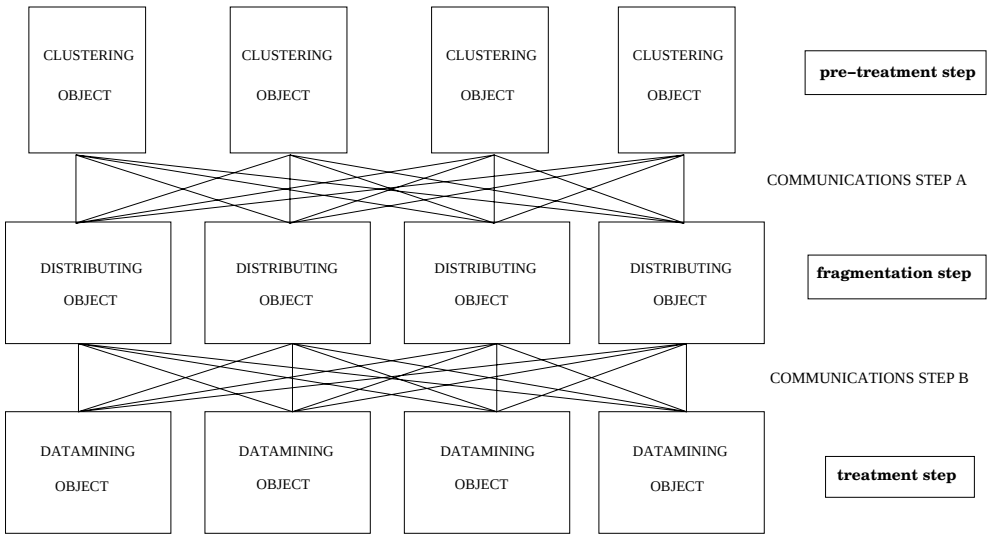


FIG. 2.2. *A Components Vision*

- The **Fragmentation step** by Distributing Objects.
- The effective **Treatment step**, the computation of association rules, by Datamining Objects.

Each stage of the general schema is described in the next section.

2.2. Successive Steps.

2.2.1. Pre-processing step. The pre-processing stage consists in a binary transformation of the database. Each attribute of the database is composed of a set of continuous values. Database attributes are classified to format the data for the problem. Since no classification exists for most of the database attributes, classes must be identified for each. A clustering algorithm (K-Means) that aims to identify groups (clusters) of values is used, those groups will be used to classify the database. This algorithm looks for groups of values:

- values in a group that are most similar,
- values in distinct groups that are least similar.

Let G_j be the j^{th} identified group; c_j, the medium value for this group; r_i, the i^{th} record of the database; and b_{ij} the bit that represents the presence of the j^{th} group for the i^{th} record. Values of a G_j group are then replaced by c_j in the database. Then for each record, r_i, of the database, D, and for each identified c_j, a b_{ij} bit can be used to represent c_j, the presence for the r_i record (see Section 3.3 for specific results on the database).

In this way, a binary database is obtained. For each attribute from the initial database, x binary columns are obtained (with only one bit equal to 1, one class for each initial attribute of a record).

2.2.2. Fragmentation step. The entire process is based on the fragmentation stage which consists in identifying groups or profiles of records. Data is then fragmented according to these profiles.

The distribution of processes with regards to profile fragmentation is related to work [6] in which association rules are computed, and groups (profiles) are then generated on the new compact representation of the data (association rules). This approach is not adapted to large databases (because of the exponential cost of association rules computation). It is proposed to approach the problem by building on the work of Lent [6]. The aim is to identify groups of similar records and then to compute association rules on these groups.

This step of data distribution will be discussed later.

2.2.3. Process step—Association rule computation. In this step of the schema, the database is fragmented in binary fragments, representing groups of records so that records in a fragment are most similar, and records in distinct fragments are least similar (the fragmentation respects clustering criteria).

The effective process could then run independently on each data fragment, using the classical sequential Apriori algorithm described in Section 1.1. Association rules are computed on local data.

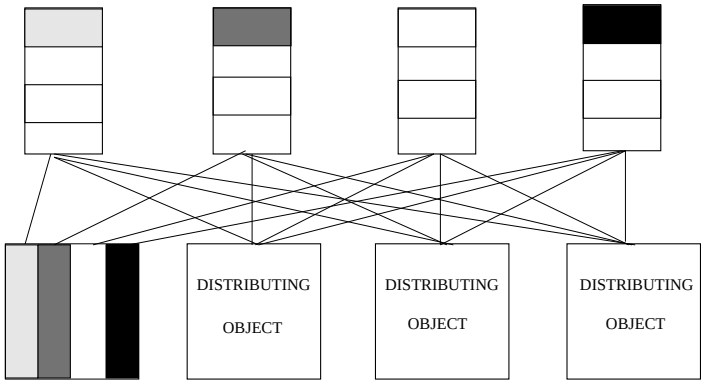


FIG. 2.3. *Communications between Clustering and Distributing Objects: Records combining.*

2.3. About Communication Evolution. Contrary to existing parallel algorithms that produce many synchronous communications between each step, the approach suggested here tends to minimize the number of communications until a quasi-zero-communication effective process.

Communications are limited to the displacement of data between each step: after pre-processing before distribution (Communications step A)) and before process by Apriori algorithm on fragments (Communications step B).

Let C_i be the i^{th} Clustering Object; D_i, the i^{th} Distributing Object; Da_i, the i^{th} Datamining Object; Fa_i, the i^{th} data vertical fragment from initial database; and Fb_{ij}, the j^{th} binary data horizontal fragment sent by C_i.

2.3.1. Data communications. Between each processing phase, the data is communicated to the following objects.

As each attribute can be treated independently, the distribution for the pre-processing step can simply be derived from the initial distribution of data (assuming that the initial distribution comes from a vertical split of the database). A C_i Clustering Object will then treat the Fa_i vertical data fragment. Therefore no communications are needed at the beginning of computation.

The first step of communications consists in communicating data from pre-processing (clustering) objects to Distributing Objects (see Figure 2.2 - Communications step A). Data has been split vertically for the pre-processing, the distribution criterion is based on entire records. The distribution is here based on an horizontal split, so data has to be recombined as shown in Figure 2.3.b. Each C_i Clustering Object will send horizontal data fragment F_{ij} to the D_j Distributing Object. D_j Distributing Object will then merge each F_{kj} fragment from Clustering Objects by a vertical recombining.

The second step of communications consists in communicating data from Distributing Objects to Datamining Objects (see Figure 2.2—Communications step B). The data distribution is provided by computations on Distributing Objects. Potentially each Distributing Object has to communicate data to each Datamining Object.

2.3.2. Pipeline. The pre-processing and distribution processes can be pipelined. Since pre-processing of each attribute of the database runs independently of every other attribute, as soon as the pre-processing of one attribute is finished, data for this attribute may be communicated to the Distributing Objects, meanwhile another attribute can be pre-treated. In this way, part of communication time can overlap with the computation.

This overlapping of communications can also take place between Distributing and Datamining Objects (see Figure 2.4).

2.3.3. Criteria for communications. The difference between synchronous and asynchronous communications, for the problem, must not be forgotten:

- synchronous communications could lead to an optimal choice for fragmentation from a global state.
- asynchronous communications allow a communication overlapping that decreases the latency time, but that does not permit an optimal split.

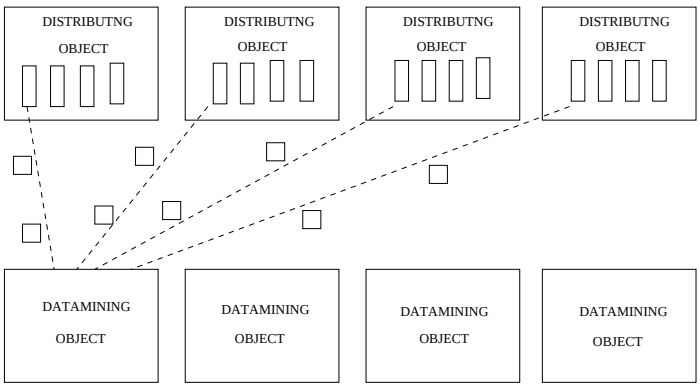


FIG. 2.4. *Pipeline between Distributing and Datamining Objects.*

Asynchronous communications are intended to be used since the purpose is to work on a NOW with irregular communication delays.

3. Data Distribution Step. The fragmentation problem consists in distributing the data in a way to keep similar data together, and least similar are split into distinct fragments. Using this kind of distribution, it could be expected that records in a fragment contain a subset of frequent items. Computation of frequent itemsets will then be less expansive .

Given n records and m items for all the databases; given n' records and m' items for a data fragment (with $n' < n$ because of fragmentation of records, and $m' < m$ as expected from profile fragmentation; distinct fragments have distinct sets of frequent items which are subsets of the whole set of items). The potential complexity (number of itemsets to be evaluated) for the process of the considered fragment (without pruning of infrequent itemsets) is then $O(n'2^{m'}) < On2^m)$.

So, the potential process complexity for K fragments is:

$$O(n'_k 2^{m'_k}) < O(n2^m) \text{ where } \sum_{k=1}^{K} n'_k = n.$$

A real decrease of global complexity could be expected. To conserve coherence of the results, data has to be distributed by identifying profiles. Then each data profile will be computed independently. The problem is that profiles are unknown.

As no global vision for the distribution of records exists, classical clustering algorithms can not be used to identify record profiles (clusters of records). The fragmentation supplied would not be optimal.

3.1. Fragmentation methods.

1. A first idea consists in using results of the pre-processing phase to distribute records. An attribute of the initial database gives X binary attributes in the binary database to be distributed. Only one of the binary attributes can be present for a record. Distributing data using the decomposition of one initial attribute and repeating this process using several initial attributes, to obtain uniform size fragments, could then be imagined.
   This proceed does not decrease the complexity of the problem. Only the attributes used for distribution become trivial. Other attributes (or items) must be computed, since attributes that classify the database do not exist (or at least are unknown).
2. A second idea consists of folding records (using binary logical functions : AND, XOR, OR...), and distributing records using the value of the obtained folding. The major difficulty is here that the local similarity between two records is lost in the folding, and two similar records may be separated in the distribution.
3. A third idea, that particular attention is given to, consists in using one (or several) pattern record(s), the same for all Distributing Objects, and to distribute data using a distance to this (these) pattern record(s).

Some pattern record(s) and a distance function (between patterns and records to distribute) should be chosen.

3.2. Fragmentation with regard to pattern records. The first parameter for the distribution step proposed consists in the choice of **pattern record(s)**.

Initially, a pattern record was randomly generated. This did not match with the database to be managed. The solution was not satisfying because of the random effect: the pattern did not represent the database structure.

Next, a pattern record was randomly elected in the database (one record is elected as a referential pattern record). Results were then relative to the database being worked on. The furnished distance values were acceptable with regards to other parameters. But this solution is not really satisfying because of the random choice of the pattern; a particular record may be chosen (as a pattern) that will clearly influence the split. A possible variation is to use a limited number of referential pattern records or to accept only records which agree with some quality function.

Work is now oriented on the computation of a pattern record that matches with the database (a kind of medium record for the database which conserve records structure), using all the information from pre-processing phase. For example, during pre-processing phase it is possible to obtain the proportion of bits equal to 1 for a binary attribute.

The second important parameter in the schema is the **distance function** used to distribute data according to distance from the pattern.

The first distance function that was included in the schema was the Hamming distance function. It counts equality of bits between the pattern and each record to be affected to a fragment. The Hamming distance function seems to be an obvious choice since a binary database is used at this stage of the process, but is too simple for the problem. Other distance and dissimilarity functions on binary data inspired from Jaccard, Salton and Cosine similarities were then used, as well as some using logical function properties. But these experimentations did not produce expected results. The obtained dissimilarity values were no more suited to the problem than those obtained with Hamming distance.

The problem with these functions is that they do not distinguish between a match of items and a match of bits. As the aim is to obtain binary fragments in which records are most similar in term of identical items (bits equal to 1), particular attention should be given to this similarity of items (similarity of bits 1).

A dissimilarity function inspired from bioinformatic work for sequence alignment has been constructed. It gives a particular penalty to dissimilarity of bits, and gives a gain to similarity of items (bits 1). In this way, one dissimilarity of bits is accepted for x similarities of items. Dissimilarity values obtained in this way are much appropriate to the application.

As work is actually directed on the use of a pattern that describe the database (a medium record for the database), the dissimilarity function may be adapted to this kind of pattern.

Computation of patterns that match with the database (using information from pre-processing) could be done in two ways:

- A binary pattern record with bit value of the majority in the records for each attribute (or item).
- A float pattern with proportion of 1 in the records for each attribute (or item).

For the first pattern (the binary one), binary distance functions exposed before could be used.

For the second pattern (the float pattern), the following distance function could be used: for each attribute (or item) the absolute float value between proportion rate (float in the pattern) and bit value is measured; then float values on each attribute are merged using Euclid's method or summation methods.

4. Experiments.

4.1. Experiment Environment. Experiments were realized on real data, a medical database composed of 182 continuous attributes (before pre-processing phase) and 30.000 records. After the clustering phase, we obtained 480 binary attributes (or items). We have increased the width of the database but many rare binary items were pruned at the beginning of frequent itemsets generation. The complexity of the database for the considered problem comes from its large number of attributes.

Programs were realized using Java RMI, on a linux heterogeneous environment, pipeline mechanisms for the distinct phases were implemented.

4.2. First Experiment. During the tests, the computation of frequent itemsets was stopped after frequent 1-itemsets generation (because of the complexity of this generation for next steps).

Data fragments quality was evaluated with regards to criteria for a good split :

- most similar records in the same fragment;
- less similar records in distinct fragments.

A complete evaluation was realized for the Hamming distance function (using a randomly chosen record from the database as pattern record), on which pertinence of the distribution was computed using criteria exposed above.

Observations on the distinct steps during tests and on criteria of distribution serves to highlight some difficiencies in our distance function (Hamming distance), that is intended to be corrected in the current works.

From a human observation of obtained fragments (observations of binary tables), the distribution using Hamming distance does not fulfill the distribution criteria; the distance function is used to compute a mathematical evaluation of those distribution criteria that is inappropriate (it is not simple to look for clustering criteria on binary data).

In addition, a dramatic imbalance in partitions was achieved.

A comparison of generated frequent itemsets is not relevant since the generation of frequent itemsets was stopped after the first stage (1-itemsets).

Other experiments were realized using distinct binary distance functions, with distinct binary pattern records (a randomly generated pattern record, a randomly chosen pattern record, a pattern record with the bit of the majority for each attribute, and a pattern of proportion for each attribute (see Section 3.2).

A real problem of imbalance was due to the arbitrarily choice of intervals for the distribution steps (see Table 4.1). For ten arbitrary fixed intervals, a lot of empty or almost empty fragments were obtained. For some distance functions, such as Jaccard, for example, only one fragment was obtained independently of the pattern record used. For most of the distance functions, only four or five fragments where obtained when ten were expected.

4.3. Second Experiment.

- Because of the complexity of the process of computing the whole set of frequent itemsets and comparing the distributed results to a global (not distributed) process of data, the distinct distance functions and pattern records were tested, first on synthetic data, next on a small fragment of the medical database (20 continuous attributes, 600 records; 23 binary items were obtained after pruning infrequent).
  Then frequent itemsets could be computed on the entire dataset to compare these results to those obtained from the distinct distributed solutions (distinct from the perspective of distance functions and pattern records used).
- In the asynchronous schema, distance intervals are used to distribute data (to simulate a clustering method on distance values), but these intervals can not be fixed arbitrary as done in the first experimentation. Arbitrary values of intervals bring a dramatic imbalance in distribution, in particular, at the end, all records could be in the same fragment or in only four or five fragments (see Table 4.1).
  Intervals should be adjusted to the database using pre-processing information. To bypass this problem, a synchronization method was used during the distribution stage. A clustering algorithm was used on the obtained distance values (distance values to the pattern record(s)). Clusters of values were obtained, and then data was distributed according to those clusters of distance values (see Table 4.2).

Evaluation of a distribution

Multiple criteria are needed to evaluate a good method. Relevance of methods could be evaluated on three points:

1. The rate of preservation of frequent itemsets to maximize (or rate of loss to minimize). (loss of frequent itemsets: globally frequent and not frequent by distributed process).
2. The rate of false positive frequent itemsets to be minimized (false positive frequent itemsets: frequent in a distributed process and not globally frequent).
3. The complexity of process on each workstation (addition of complexities in a distributed schema must be less than the global complexity).

TABLE 4.1
Rate of distribution of records for some distance functions using intervals

Distance function	Rate of distribution of records (only not empty fragments appear in the table) (%)
Binary randomly chosen pattern record	
Hamming	1,7 16,5 33,5 37,7 9,8 0,8
Hamming variation (Items2)	1,3 7,8 28,8 43,3 16,5 2 0,17
Bioinformatic (2)	1,3 7,8 28,8 43,3 16,5 2 0,17
Salton	0,7 3,5 3,3 13,5 66,3 12,7
Cosinus	0,7 1,8 2,17 22,7 22,8 38 11,8
Jaccard	0,17 99,8
Binary randomly generated pattern record	
Hamming	1,7 1,7 10,8 50,17 37,17
Hamming variation (Items2)	6,7 68,17 25 0,17
Bioinformatic (2)	3,17 30,7 49,17 0,16 1
Salton	2,3 97,7
Cosinus	0,3 13,8 85,8
Jaccard	99,7
Bit for majority of records	
Hamming	7,8 45,17 39,3 6,5 1,17
Hamming variation (Items2)	7,8 45,17 38,3 6,8 1,7 0,17
Bioinformatic (2)	7,8 45,17 38,3 6,8 1,7 0,17
Salton	47,3 28,5 8,8 15,3
Cosinus	5,7 10,17 9,7 42,17 19,8 12,3 0,17
Jaccard	100
Float pattern of proportion	
Summation	8 85 3,3 1,7
Euclid	5,7 46 39,7 6,8 1,8

TABLE 4.2
Rate of distribution of records using clustering of distance values

Distance function	Number of identified clusters	Rate of distribution of records (only not empty fragments appear in the table) (%)
Bit for majority of records		
Hamming	5	14,8 17 10,7 10,3 3,5
Hamming variation (Items2)	6	19,3 16,3 26,17 20,7 9,7 7,8
Bioinformatic (2)	7	19,3 16,3 13,8 12,3 20,7 9,7 7,8
Salton	3	42,67 35,67 19,67
Cosinus	5	52,67 17 21,5 4,5 4,33
Jaccard	2	80,67 19,33
Float pattern of proportion		
Summation	3	19,33 38 42,67
Euclid	4	19,33 12,33 28,8 39,5

Results

For the distance functions that give a sufficient discrimination of records (a sufficient numbers of clusters

and a sufficient number of records in each cluster), a rate of preservation of frequent itemsets of 100 %, and a rate of false positive frequent itemsets between 15% and 20% (depending on fragment) of distributed generated frequent itemsets were reached.

Addition of complexities on each fragment is less than the global complexity because of non frequent items on fragments that permit to decrease the number of itemsets to be computed, and because of obvious items not included in the process (see Observations above). Less itemsets are computed on each fragment than in a global process and addition of complexities from each fragment is less than the global complexity.

Distinct Problems

1. Problems from a non uniform data distribution: Some distance functions do not permit the database to be broken into a sufficient number of distinct fragments (profiles), some like Jaccard, Salton, Summation can be immediately rejected on this basis(see Table 4.2).
2. Problems in the evaluation: Generated frequent itemsets are compared in a global process and in a distributed process, by merging results from fragments without paying attention to support values. The threshold for frequency in the distributed solution is relative to the size of fragments (in particular, little fragments give problems of complexity and should be treated separately). A good evaluation could not be made without a corrective method (see Section 2.1), that could consist in adjusting threshold support to local data using pre-processing information.

Observations

If a threshold of obviousness is used, beyond which an item could be considered to appear in all records, many items can be pushed aside, this will result in a decrease in the complexity of the computation. Those items will then be replaced at the end of the process, or be given separately from results.

In the distributed experiments, more obvious items were found on fragments than in a global process. This confirms the relevance of distribution (see Table 4.3).

The complexity to which results refer takes into account these obvious items (they have been pushed aside at the beginning of process on fragments).

TABLE 4.3
Rate of obvious items (threshold of obviousness : 95%)

Distance function	Rate of obvious items (only not empty fragments appear in the table)
Bit for majority of records	
Globally obvious	26,09
Hamming	43,48 56,52 26,09 34,8 4,3
Hamming variation (Items2)	69,56 43,48 52,17 34,8 26,09 4,3
Bioinformatic (2)	69,56 43,48 60,87 52,17 34,8 26,09 4,3
Salton	26,09 52,17 8,7
Cosinus	56,52 43,48 34,8 26,09 4,3
Jaccard	26,09 30,4
Float pattern of proportion	
Summation	69,56 43,48 17,4
Euclid	69,56 69,56 43,48 17,4

4.4. Future Works.

- We intend to compute interval values that could bring a good heuristic for this clustering method with regard to distance values. Maybe a clustering algorithm could be computed on a partition of the database. Work is in progress for this part.
- Corrective method : the support threshold on each site has to be adjusted to data on this site (this could clearly come from the beginning of computation on site by evaluating 1-itemset support, and with a comparison to global support (information that we get from pre-processing).

This corrective method and the arrangement of distributed threshold support would permit, we hope, a decrease complexity on some sites. It might be not be necessary to make a distinction for sites with too few records.

Tests show that it is necessary to draw a parallel between the relevance of frequent itemsets and the distribution of data (the size of distinct fragments). It will be necessary to test other databases to be sure that orientations derived from the observations made on the medical database are not specific to these data.

The schema uses many parameters (local threshold, distribution intervals, etc. . .) that should not be fixed arbitrarily. Values for those parameters have to be computed from pre-processing information.

5. Conclusion. It is necessary to remember that the schema will supply a heuristic for the problem of association rules. The distribution of data is the critical phase to obtain a good heuristic for the problem. To compute profiles without a global vision of data is of course a limitation to the process but the distribution could provide sufficient results with regard to speeding-up te process.

Many directions need to be explored, particularly for the choice of referential pattern records and the distance function. A lot of information can be obtained from the pre-processing phase and it is important to use it to distribute the data. This is the direction given to this work. It is intended to duplicate some records in the distribution. This will need a specific corrective method to take those duplications under consideration. It is also intend to combine the clustering phase and the distributing phase to avoid data communications between them.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami, *Database mining: A performance perspective,* IEEE Transactions on Knowledge and Data Engineering : Special issue on learning and discovery in knowledge-based databases 5(6):914-925 (December 1993).

[2] R. Agrawal and R. Srikant, *Fast algorithms for mining associations rules in large databases,* In Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB'94), pages 478-499 (September 1994).

[3] R. Agrawal and J.C. Shafer, *Parallel Mining of Association Rules,* IEEE Trans. on Knowledge and Data Eng., 8(6):962–969 (December 1996).

[4] D. W. Cheung, J. Han, V. T. Ng, A. Fu and Y. Fu, *A fast distributed algorithm for mining association rules,* In Proc. of the 4th Int. Conf. on Parallel and Distributed Information Systems, pages 31-42 (1996).

[5] E-H. Han and G. Karypis. Scalable Parallel Data Mining for Associations Rules, IEEE Trans. on Knowledge and Data Eng. , 2(3):337-352 (May-June 2000).

[6] B. Lent, A. Swami and J. Widom, *Clustering Association Rules,* In Proc. of the 13th Int. Conf. on Data Eng. (ICDE'97), pages 220-231 (April 1997)

[7] J. S. Park, M.-S. Chen, and P. S. Yu, *Efficient parallel data mining for association rules,* In Proc. of the 4th Int. Conf. on Information and Knowledge Management, pages 31-36 (1995).

[8] A. Savasere, E. Omiecinski and S. Navathe, *An efficient algorithm for mining association rules in large databases,* In Proc. of the 21st VLDB Int. Conf. (VLDB'95), pages 432-444 (September 1995)

[9] R. Srikant, *Fast algorithms for mining association rules and sequential patterns,* PhD thesis, University of Wisconsin (1996).

[10] Zaki, *Parallel and Distributed Association Mining: A survey,* (1999).

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: Semptember 30, 2002
Accepted: December 21, 2002

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 111–??. http://www.scpe.org
ISSN 1895-1767

THE PROBLEM OF AGENT-CLIENT COMMUNICATION ON THE INTERNET

MACIEJ GAWINECKI[¶], MINOR GORDON[†,‡], PAWEL KACZMAREK[¶] AND MARCIN PAPRZYCKI[‡,§]

Abstract.
In order for software agent technology to come to full fruition, it must be integrated in a realistic way with existing production technologies. In this paper we address one of the interesting problems of *real-world* agent integration: the interaction between agents and non-agents. The proposed solution is designed to provide non-agents (client software in particular) access to agent services, without restricting the capabilities of agents providing them.

1. Introduction. The future success of online content providers will depend on their ability to filter the mass of information available on the Web into the form that is truly useful to the individual user [26, 23, 44, 25, 2, 64]. Before filtering services can be automated and scaled, however, the World Wide Web must first be transformed to a machine-interpretable content base. One of the attempts at achieving this goal is the Semantic Web [53, 54, 55], which promises to allow computing processes to filter, separate and synthesize elements of information [14]. The major thrusts of the Semantic Web effort are the development of languages to describe content ([48, 13, 42, 62, 63]) and of software capable of interpreting this content.

Software agents are one of the key components in this latter category [54, 55]. Research in this area has long emphasized the potential of agents for intelligently interpreting semantic content, personalizing this content for the user, and acting autonomously and collaboratively to deliver it. Agents representing users are in a position to intelligently filter information from the Web in order to satisfy a user's request by consulting dynamic and static sources of semantically-described content as well as other intelligent agents. A large body of literature has developed around this role of software agents (see for example [27] and references quoted there).

As a *framework* for conceptualizing intelligent computing processes such as those required for working with the Semantic Web, software agents are close to ideal. As a *specific technology* for working with the Semantic Web data, however, agents are far from ready to assume the roles ascribed to them. Though the theoretical concepts that define the software agent paradigm have been thoroughly researched (and remain a focus of current research efforts: see [34, 15] as well as recent proceedings from multiple agent-focused conferences), technological advances to support agent-based models have usually not accompanied this research [37]. Many proponents and even standards bodies (in particular FIPA: [22]) have described detailed scenarios in which agents play critical roles, yet very few of these designs have progressed beyond the initial stage of development. Although the absence of supporting technology has certainly been one of the reasons for this gap between agent theory and practice, we believe that the lack of realistic implementations of flexible agent-based applications has also resulted in part from the attitudes of agent researchers. Most designers have either been too "realistic" and implemented systems that can hardly be called agent-based, or have made so many assumptions about the *future* of agent technology (i.e. practically every computer connected to the Internet is capable of receiving agents [22]) that their designs will almost certainly be outmoded before that state of technology exists and are impossible to implement beyond some initial working model employing *currently* existing software. Serious efforts toward finding an intermediate solution have only recently gained strength, and many of the conceptual problems pointed out by critics such as Nwana and Ndumu [37] have begun to be addressed. In addition to the basic problems of communications management and ontology agreement, Nwana and Ndumu cited the development of realistic agent-based systems as both the problem and the solution to the evolution of agent technology. This paper represents our attempt to address that problem.

With this in mind, we must first recognize that developers seeking to apply software agent technology to a specific application currently have two choices: they can either wait for the ideal technology to come along and facilitate the implementation process, or make do with existing agent technology, knowing that they will have to work around it as much as work with it. While the former approach certainly has some currency for those whose goal is to develop industrial-strength applications and therefore prefer to use well-established technology, we focus on the latter in our research. By choosing this approach we acknowledge that any application we develop with existing, rudimentary technology will almost inevitably be outmoded in the future by more elegant

[¶]Department of Mathematics and Computer Science, Adam Mickiewicz University, Poznan, Poland
[†]Computer Science Department, Technical University of Berlin, Berlin, Germany
[‡]Computer Science Department, Oklahoma State University, Tulsa, OK 74106, USA
[§]Computer Science, SWPS, Warszawa, Poland

solutions. Our hope is that in the process of developing such an application we may help shape the "ideals" of software agent technology itself, so that future agent software developers will find it easier to work in realistic contexts.

The application we are in the process of developing is an agent-based system supporting the needs of travelers. In this paper we address one of the interesting problems that has materialized during the implementation phase: the problem of software agents communicating with non-agents, the latter being the clients of the system. The remaining parts of the paper are organized as follows. In the next section we briefly describe the system under development and follow with a discussion of the role and requirements for agents communicating with a heterogeneous, functionally-limited client base. In the subsequent section we analyze in detail possible approaches to facilitating agent-client interactions. In the next two sections we introduce and discuss the proposed solution.

2. Internet-based Travel Support Systems—General Considerations. When considering existing travel support systems such as Expedia, Orbitz, Travelocity, et al., one tends to focus on the content aspects of the system—the travel services and information these sites provide—and not the technology that makes them work. The technology is (and should be) transparent. Behind the scenes these travel support systems are enormously complex, and adding further levels of intelligence / user friendliness will only make them more complicated (for more details of current trends in development of intelligent travel support systems see [28] and references enclosed there). From this perspective software agents represent an excellent means of breaking apart complex systems into components (agents), each with its own "knowledge" and "goals", thus allowing developers to better address the complexity of the system [32]. In addition, travel support (browsing, booking, etc.) is an ideal candidate for agent-based implementation because the services offered by Internet-based travel support systems are those we associate with another kind of an agent—the human travel agent. An agent or agents can be designed to directly adapt the role and functions of the human travel agent by following our conventional intuition of an "agent" as someone/something who represents us, usually in a particular domain such as travel (real estate, insurance, etc.). In this sense, a software agent-driven travel support system can work like a real "travel agency", with individual "travel agents" in the system serving individual users, just as a human travel agent would serve an individual in person. We would also like to incorporate other perspectives on software agents, namely, the conception of agents as autonomous, intelligent entities, which has arisen from the field of artificial intelligence. It is our belief that in order to imitate the role of a human travel agent with our agent-based travel support system, we must start with a software agent assuming the travel agent role and design the other agents to support this "travel agent". In fact, we call this central agent the "personal agent," because it represents the travel support system to the individual user; there is one personal agent running for every user accessing the system. The other agents who support the personal agent have been developed as functional components of a distributed, complex system. These secondary agents support the personal agent in fulfilling the user's requests: researching available options, developing travel plans and choices to fit the customer's preferences, booking, etc. The personal agent is responsible for interacting with the user and executing various aspects of her request by directing other agents in the system.

2.1. System Under Development—High Level Overview. Our work on development of an agent-based travel support system was initiated in 2001 and first described in [23, 43]. This work led to the conceptualization of a complete framework presented in [2]. The processes involved in personalization of information delivery (independent of the implementation) were discussed in [25]. Knowledge management in its conceptual and agent-related implementation aspects was considered in [26, 29, 23, 64]. Our system has been designed to imitate the role of a human travel agent in serving individual travelers: presenting a wide selection of travel choices, planning itineraries and representing the customer to airlines, hotels and vehicle rental agencies as well as other, more information-oriented services such as displaying the hours of operation of museums and historical sites or the locations of nearby restaurants. In order to provide these services the system accumulates semantically-demarcated travel-related information collected from the Internet, which is cross-referenced according to multiple classification schemes (geospatial, ontological, method of access, etc.).

Customers connect to our system using Internet-enabled devices—web browsers, PDAs, WAP phones and others—and request travel-related information. The initial response to a request is prepared from the information stored in the system and filtered to match the user's personal preferences. The system also attempts to sell additional services to the user through targeted advertisements [25]. In the process of interacting with the system, customer's queries may be refined as more specific choices are presented. All of the details of customer-

system interactions are stored in a user behavior database for later data mining [2, 24]. The complete system framework is specified in terms of software agents, with separate agents performing all of the above-described functions [32].

One of our most important design criteria was to make our travel support system accessible not only to the general Internet audience (typically accessing services through a desktop-based web browser) but also to travelers accessing services through mobile devices such as laptops, PDAs and cell phones. Each of these technologies has a certain set of capabilities and limitations that must be considered in providing access to the different devices. This is the fundamental problem for software agents interacting with non-agents (in this case, client devices).

Because travel support is such an ideal scenario for applying agent technology, there have been a number of agent-related projects in this domain; unfortunately, most of these projects have either been very limited in scope [36, 57, 58] or have never left the initial development stages (a partial list of the projects that have never been completed can be found in [43], while others can be easily located through Internet searching). Unlike most of these projects, our essential goal is not to prove the elegance of design with software agents, but to demonstrate that agent technology, as it exists *today*, can actually be used in a real system. This is demonstrated by the fact that the problem of agent-non-agent interaction only arises when agent systems are actually implemented, and thus it has been largely ignored or downplayed by other researchers [22, 37].

3. Clients of the Travel Support System. In order to develop device-independent support for customer-system communication we have to start from the client side, as it is the diversity of possible clients that is the source of the agent-client interaction problem. Despite the diversity of device types the system must support, we can still assume that all interaction between devices (clients) and the system on the Web will be conducted via the HTTP protocol. Only the *content* that is delivered over HTTP that will differ from client to client. We can divide clients into two broad classes, based on the type of content they expect:

- *Non-interactive clients*, which include but are not limited to web service clients (using SOAP [56]), geospatial clients [39, 40], travel industry systems [41] and external agents [1].
- *Interactive clients*, which include among others conventional web browsers (interpreting HTML) and browsers on personal devices (interpreting HTML or WML) as well as interactive applications that utilize the services of our system.

In this paper we will focus on browsing clients in the latter class. Though the class of interactive clients described above also includes applications (Flash-based, Java applet-based, etc.), these are not the primary client base of the system, and furthermore, we believe that users should not have to (and may not be allowed to) download any special software in order to interact with our system (more on this below). More precisely, if a user can and is willing to load a special applet that will allow her computer to accept mobile agents, then the problems that are of interest to us can be solved in a different way, or cease to exist completely. Henceforth we will refer to browser-like clients (desktop and mobile) as simply "clients". However, the solution proposed here can be naturally and easily extended to include, among others, scenarios involving software agents entering user devices. Let us now consider the basic limitations define the possible solution space and the existing capabilities that we can exploit for our system.

3.1. Client Limitations and Capabilities. Although the HTTP protocol itself is relatively standard across browsing clients, browsers are often limited in their ability to render content sent from a server via HTTP (whether in HTML or WML, and in particular XML). Therefore we cannot impose many requirements on browsing capabilities when considering an audience as large as travelers on the Internet. This automatically precludes platform-specific, device-specific, browser-specific or otherwise specialized technologies on the client side such as Flash movies or Java applets, which are still largely inaccessible to non-desktop-based browsers. Some users may also be unable (personally) or not allowed (administratively) to modify settings or install programs on their computers, while others may reject anything they consider "foreign". Furthermore, despite the widespread distribution of XML parsing technology in web browsers and runtime environments such as Java, we still cannot consistently rely on the end user's software to interpret or transform content beyond the standard markup languages such as HTML or WML. We must consider the client as "dumb" as possible, so we don't exclude clients that actually are so restricted.

On the other hand, for a web-based travel site to offer useful content and services it must still require a minimum subset of client-side interpretative capabilities for interactive access, client limitations notwithstanding:

- Client devices / software must be able to display some flavor of *markup* (HTML, WML, XHTML, etc.)
- Clients must be able to *connect one-way* to services over the Internet using HTTP.

It has to be stressed that the above-described capabilities represent the required minimum. This being the case, the system may still be able to adapt to discovered capabilities of the client: for clients with only minimum interpretative function (e.g. the typical browsers we are focusing on), the server system is responsible for most of the interaction and interface with the user, while more sophisticated client applications may simply call upon the system's functions directly (as is the case with fully-interactive applications as well as the non-interactive clients specified previously). In this paper we concentrate on providing a *minimum subset* of markup (HTML and WML) and protocol (HTTP) we can assume **all** browsing clients can handle, and leave the extended capabilities of the travel support system for the future.

3.2. Client Agnosticism. One of the strengths of the web's client-server architecture is the agnosticism of clients with regard to server technology. As long as a web server provides interpretable content (markup, images, etc.) over HTTP, the web browser need never be aware of how this content is produced. The same web page may be derived from static files, querying a database or sending a request message to an agent across the network. The client only sees the end result—a web page. We will take advantage of this client agnosticism by designing a server system that *appears* to work like a normal web server but handles requests much differently on the server side.

4. Communicating Agents of the System. This far we have constructed the following general picture of our work. We are in the process of developing of an agent based travel support system in which software agents will have to communicate with customers accessing the system using devices that we must assume have only very limited capabilities. Let us now consider two basic but unrealistic possibilities for agent-client communication within this system.

4.1. Agents Accessing the Client Devices. In the last section we arrived at the basic assumption that our agent-based travel support system should be web-based and operate within the conventional web browser—web server infrastructure. Unfortunately, this assumption contradicts one of the central ideals of software agent research: agents should be everywhere. The user should have an agent on her desktop—either sent by the server or already present as the user's personal assistant—and this agent should be responsible for interacting with the web, in lieu of the conventional web browser. Despite the fact that this vision is still only an ideal, many agent-related software projects assume that an agent is/will be present on the user's system. The reality is that agent platforms generally require a host environment ("agency", "platform", "marketplace") to support agent execution on the local system. Furthermore, there are only a few projects that are aimed at supporting agents entering mobile devices [33]. The presence of such a specialized environment on a range of client devices is obviously not a realistic assumption.

4.2. Agents Exchanging Messages with Clients. Here we consider a scenario in which a conventional browsing device (desktop- or device-based) establishes an HTTP client-server connection directly with a software agent, as illustrated in Figure 4.1.

4.2.1. FIPA. After some years of uncertainty and in-fighting between different organizations and companies, the Foundation for Intelligent Physical Agents (FIPA) [15] has emerged as the de facto standard bearer for the agent community. FIPA has defined standards and specifications for inter-agent communication—the Agent Communication Language [16]—as well as guidelines for agent platforms. These standards are gradually being adopted by the agent community at large, and have been the basis for several general-purpose agent platforms [30, 7, 8]. One of the more popular of these—the Java Agent Development Environment (JADE)—has been widely deployed because of its close conformance to FIPA specifications and the resulting endorsement by large-scale projects such as the AgentCities network [1], which have a clear interest in platform interoperability. JADE is our platform of choice for the travel support system.

Unfortunately, FIPA and the developers of FIPA-compliant platforms such as JADE have concentrated almost entirely on the development of inter-agent and inter-platform protocols and languages such as ACL, with only marginal regard for the possibility of non-agents (i.e. clients such as ours) interacting with FIPA-compliant agents. Even the inter-agent communication mechanisms based on web standard protocols and formats (e.g. the HTTP Message Transport Protocol [19] and the XML encoding for ACL [17]) are designed only for transporting ACL messages between agents. Non-agents, even if they can "speak" HTTP, cannot interpret the contents of ACL messages sent from agents without special software. The situation is further

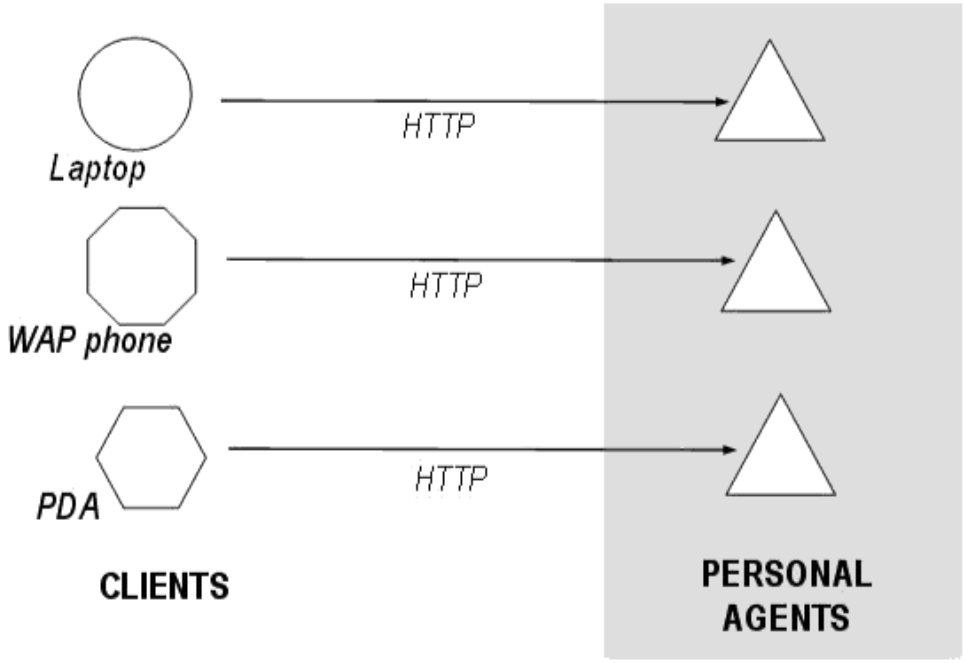


FIG. 4.1. *Client-agent link*

complicated by the indirection imposed by the software agent platforms; agents may address each other within the platform via a FIPA-specified addressing schema, but are shielded from the larger network by the platform environment. Software agents within the platform cannot be directly addressed via the standard IP, port scheme, and thus they cannot be reached directly by client devices over HTTP.

4.2.2. Agents as Servers? Without speaking a language clients can interpret (i. e. HTML or WML) and without being able to directly travel to the desktop or to access the personal device, the agent is "cut off" from the user. The ideal of clients connecting directly to agents is also a dead end. Finally, agents (as they exist now) were not designed to be servers. An agent that remained in a known location and only responded to requests would not really be an agent by any of the existing definitions, and enforcing these limitations on agent software (existing or specially-designed) would mitigate many of the reasons for using agent technology in lieu of simple daemons in the first place. In addition, an agent that internally supported the full HTTP protocol as well as any markups the system is expected to support (HTML, WML, or XML) would be very complex and limited in its mobility and distribution.

In summary, as we have just argued, neither of the basic scenarios (agents entering the device or agents acting as servers), can solve the problem of non-agent-agent communication. Rather, we have to focus on the strengths of agent software, and let other technologies make up for its weaknesses.

5. N-Tier Systems. The conventional n-tier architecture is designed to separate user interfaces from business logic and business logic from the resources it employs, so that each tier is encapsulated and need not be aware of the operations of the others. It is obvious that the n-tier approach applies to our problem. Conceptually, the agent infrastructure becomes the middleware [50], allowing other components of the system to handle the problem of communicating with the user and to interact with back-end resources. A design based on agents acting as middleware is illustrated in Figure 5.1.

Multi-agent systems are well suited for encoding middle-tier business logic. In such a system agents should be allowed to "pretend" that they are actually interacting with and representing the user directly, i.e. on the abstract, conceptual level we want the situation depicted in Figure 4.1 while on the implementation level we will actually have the situation illustrated in Figure 5.1. This situation is actually fortuitous: a personal agent on the server can take advantage of the server environment while still being geared toward serving the individual (personalizing and localizing content derived from the system [2, 25]). There are also additional advantages of

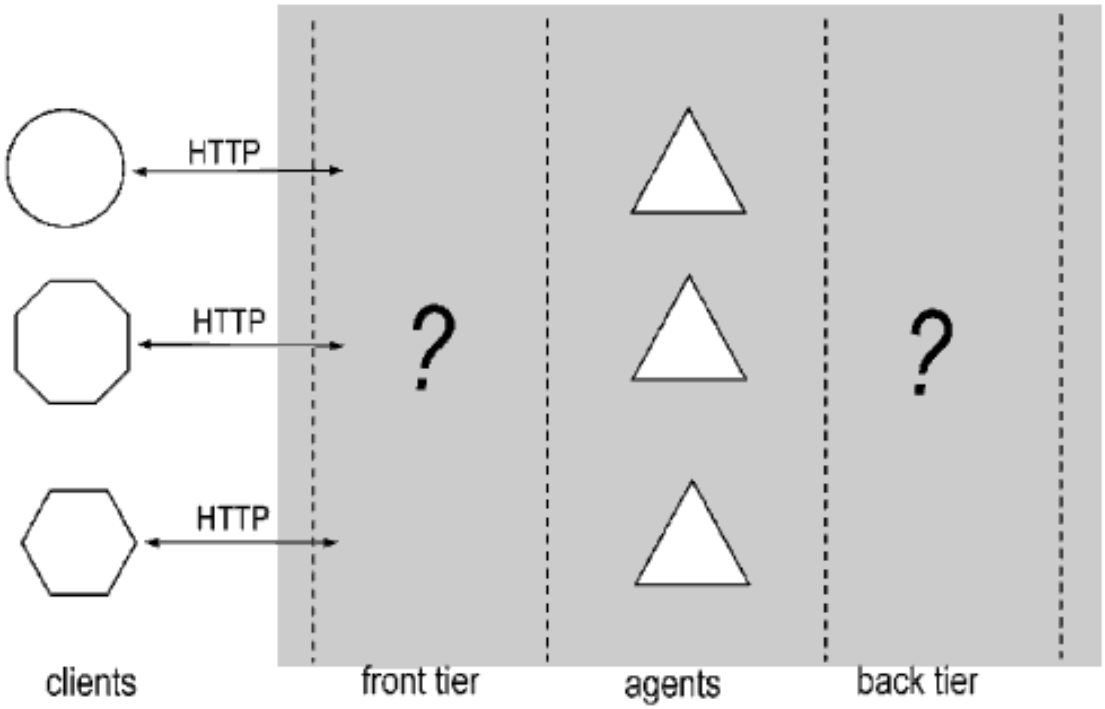


FIG. 5.1. *Agents as middleware; clients reside on the Internet; the remaining tiers reside on the server*

agents being located on the server and acting as one of the layers of the n-tier system:

- *Processing power*: agents can either reside on large servers or be spread out on to a distributed system; each has its own advantages and disadvantages, but more importantly, neither relies on the user's system or bandwidth.
- *Mobility*: an agent could be anywhere on the network, as long as it stays in touch with the server system.
- *Access to back-tier resources*: databases and other information sources can be directly accessed by agents or through other agents in closer proximity to the source.
- *Uniformity*: all agents can be exactly the same as they have been separated from the multiplicity of "languages" spoken by clients.

We would like to construct an n-tier architecture that supports the transparent integration of a general agent platform such as JADE into an otherwise conventional enterprise application, which can then take advantage of the strengths of agents (both distributive and intelligence aspects) while retaining the interoperability of existing technologies such as web servers. This relieves the agents of the problems of communicating with non-agents while still allowing them to be the real intelligence of the application—in the middle. Fortunately, the n-tier approach offers a great deal of flexibility in dividing the functions of the different tiers. Some developers choose to make the presentation layer as "dumb" as possible, even going so far as to have the middle layer output almost-presentable content for delivery to the user. Others focus on the end user functions, and make the middleware only a thin layer above the back-end resources on the network (both approaches are illustrated in Figure 5.2.) Of course these distinctions and the n-tier approach itself largely excludes pure agent-based designs, which are founded on the assumption that there is no need for intermediaries because agents will be everywhere.

5.1. Agents on the Front End. One way to integrate agents into an n-tier server architecture and allow clients to access them over the Internet is to actually embed agents within a web server, tightly binding the front (presentation) and middle (logic) tiers of the system. This approach was taken by the aZIMAS system, which uses web server modules (for Apache or IIS) to route messages with predefined HTTP parameters to agents

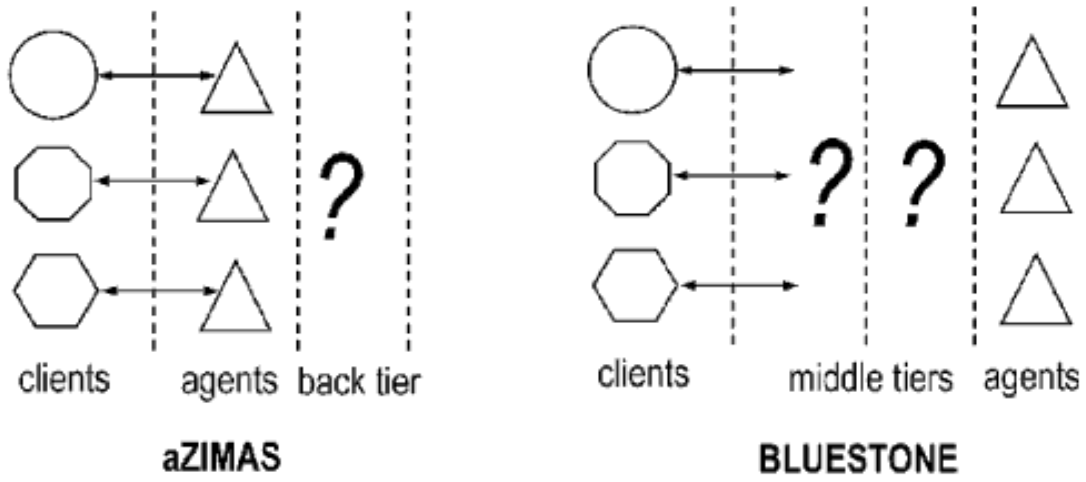


FIG. 5.2. *aZIMAS vs. Bluestone*

running in the web server's process space [3]. This is very efficient approach to the agent-client communication problem, as it integrates agents tightly into an existing server structure with a minimum of customization. Unfortunately, embedding agents in the server required the aZIMAS developers to create an entirely new agent platform to work with the system. From our perspective this is not a true solution to the problem, as it bypasses the problem of linking current agent platforms (such as those that comply with the FIPA standard).

5.2. Agents on the Back End. The opposite approach places agents on the back end of an n-tier system, treating them as service providers, in much the same way that legacy applications are exposed as web services—by representing their functions through a standard interface, and brokering requests for and responses from the functions through a middle tier. This was the approach taken by Hewlett-Packard's Bluestone middleware system [5, 9], which allows HTTP, SMTP and other client types to communicate (bidirectionally) with agents through an intermediary (the Universal Listening Framework) [6]. A service broker (the Universal Business Service) negotiates service requests from clients with agents and other service providers. After the merger of HP and Compaq, the Bluestone project was cancelled and agent-related development in the context of application services became the BlueJADE project, which similarly attempted to integrate JADE agents into an application server (JBoss) [4]. One disadvantage of using agents as service providers is that it reduces them to mere request handlers, operating through a generic interface. This limits much of the usefulness of agents, as it ties them too closely into the architecture.

6. Agents as Middleware. For the implementation of our travel support system, we employ a general agent platform (JADE) for serving clients over the Internet, attempting to incorporate the best aspects of both of the extreme approaches described above, taking advantage of agents' strengths, and minimizing exposure of their weaknesses.

In designing our system, we started from the vision of clients connecting directly to personal agents (one client per agent) depicted in Figure 4.1 with the goal of making the layer or layers between clients and agents as transparent as possible—such that conceptually we end up with Figure 5.1 while actually implementing Figure 5.2. To this end we insert a limited number of intermediaries between the client and the personal agent (the HTTP server and the *Proxy agent* in Figure 6.1). These intermediaries pass requests and responses between the user's client software (linked to the system over the Internet) and the personal that is responsible for serving that user, transforming the communications to appropriate formats as necessary. Let us now describe in more detail the processes involved in passing user query from the device to the database and the results back to the user device.

6.1. Proposed Solution. An intermediary is required to communicate with clients of various types, in order to shield/abstract the personal agent and the rest of the system from low-level client protocols (e.g.

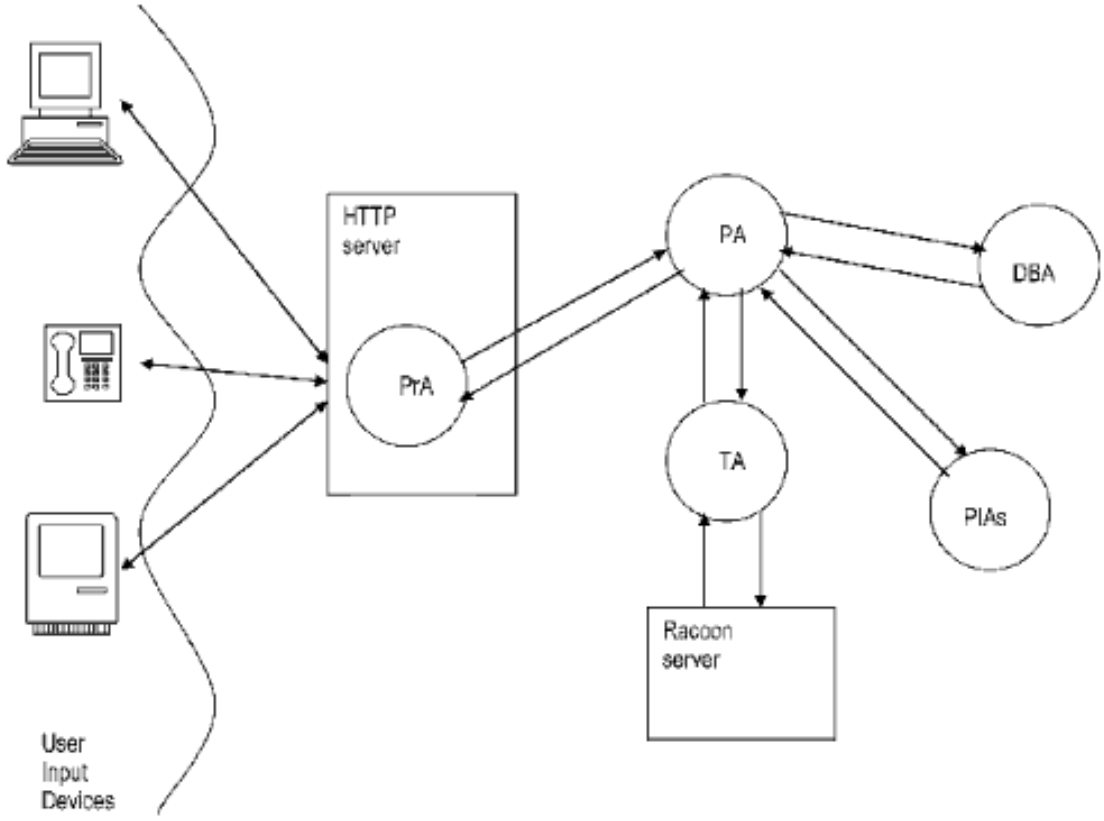


FIG. 6.1. *Client side of the system, information flow details.*

HTTP, WAP, etc.). In Figure 6.1 we represent only one *Proxy agent* that is to handle HTML-based requests and responses, while each additional protocol would be handled by a separate *Proxy agent*. We assume that requests from the system are from a specific device arrive in the form of HTTP POSTs or GETs, which are usually initiated by filling in predefined forms (see the next Section and Figures 6.2 and 6.3) and pushing a "submit" button. Within the HTTP server a *Proxy agent* receives HTTP requests from all devices utilized by users of the system, transforms them (see below) and forwards as ACL messages to the specific user's personal agent. The type of client device (which indicates the necessary transformations) can be detected implicitly (e.g. from request headers) or explicitly (from URL tokens) and stored in server state.

One should remember that HTTP is a synchronous protocol and agent-based communication is asynchronous. Thus each response from an agent-based environment needs to be matched with an appropriate HTTP request. The proxy agent must keep the HTTP connection alive until final response from the personal agent is returned. Detailed mechanics of servicing a given user-request is as follows:

1. proxy agent receives a request over HTTP,
2. a new thread is started to process the request; the request is given the unique identification ID; this ID includes the I/O device type information,
3. the thread forwards the request to the personal agent via an ACL message; from this point all messages relevant to this request must be accompanied with that ID,
4. the thread suspends,
5. the personal agent processes the request and sends an ACL response through a transformation infrastructure (e.g. to transform the response into HTML, WML, etc.)
6. the proxy agent notifies the suspended thread that the response is ready, and
7. the resumed thread sends back the response (as an HTTP body) to the user device.

When the proxy agent receives the user's query request (HTTP POST or GET), it extracts the content of the query from the CGI query string, transforms the query into an ACL message, and sends this message to the personal agent. For example, the contents of the message might be:

```
page?formvar1=val1&formvar2=val2
```

(see the next Section for a detailed example). The personal agent receives this message and forwards it to the database (*model*) agent as well as the *interaction logging* agent (omitted in Figure 6.1). The role of this latter agent is to store, for further processing and mining, information about all interactions between users and the system [2, 25]. Further discussion of this part of the system are outside of the scope of this paper. The database agent extracts the query from the ACL message and transforms it into the appropriate database query language. In our system this database returns a set of RDF [48] triples that define travel objects, which are defined with formal ontologies. The set of RDF triples are serialized and packed into an ACL message and forwarded by the database agent to the personal agent. The personal agent may elect to further expand/reduce the result set by directing a set of personalization agents to add/remove triples before the set is "complete". The user-ready response set is then returned to the personal agent for the task of transforming the RDF triples into a form that can be displayed on a user device. The personal agent delegates this operation to a *transformation agent*, which in our system is based on the Racoon server [47]. Racoon uses the RxPath language (analogous to XPath 1.0 in terms of syntax and behavior) and RxSLT stylesheets (analogous to XSLT) to transform RDF-demarcated data into displayable forms such as HTML/XML, in a manner similar to that of Apache Cocoon [12]. We have defined a separate set of RxSLT stylesheets for each type of input device we wish to support. After applying the appropriate transformations the transformation agents sends the resulting HTML/WML back to the personal agent, which forwards the displayable content back to the proxy agent for transmission to the user.

6.2. Example scenario. Here a simple example scenario should serve to illustrate the process depicted above. We assume that a user has successfully logged in to the system and established interaction with a personal agent. For this scenario the data consists of sets of RDF triples with elements from a restaurant ontology [11]:

```
:Poland_ZP_Swinoujscie_Albatros_Klub_Nocny1051910264
        a res:Restaurant;
    res:title "Albatros, Klub Nocny";
    loc:streetAddress "ul. Zeromskiego 1";
    loc:city "Swinoujscie";
    loc:country "Poland";
    loc:phone "+48 (91) 321 18 66";
    loc:state "ZP";
    loc:zip "72-600";
    res:accepts mon:AmericanExpressCard,
          mon:DinersClubCard,
          mon:JCBCard,
          mon:MasterCardEuroCard,
          mon:VisaCard,
          mon:DebitCard;
    res:alcohol res:FullBar;
    res:cuisine res:BarPubBreweryCuisine;
    res:hours "24h";
    res:locationPath "Poland/ZP/Swinoujscie";
    res:parsedHours "0-24|0-24|0-24|0-24|0-24|0-24|0-24".
```

In order to query the restaurants in the database the user fills in a form defining criteria of a restaurant search and clicks the `<query>` button. An example of the form used in our system is shown in Figure 6.2.

Here the user is looking for a pub in the city of Swinoujscie. The submitted CGI query string for this request is as follows

```
http://www.agentlab.net/restuarant/page?action=getdata&alcohol=
      FullBar&cuisine=BarPubBreweryCuisine&city=Swinoujscie
```

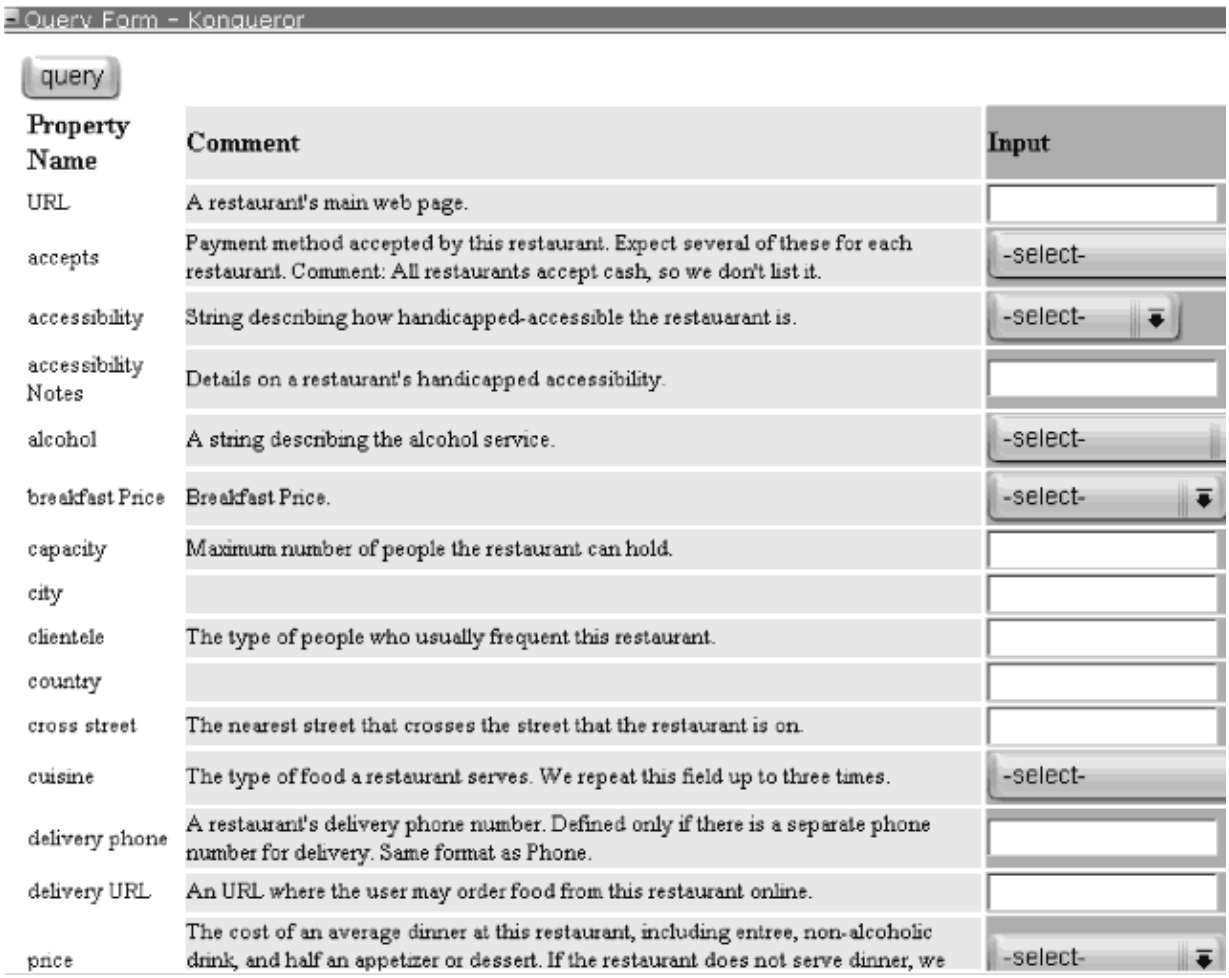


FIG. 6.2. *Input form for the system*

The proxy agent that receives the HTTP requests maps the variables of the CGI query string to the temporary form `[[alcohol{{FullBar[[cuisine{{BarPubBrewery[[city{{Swinoujscie)`, and packs this string into an ACL message, which is sent to the database agent via the personal agent (as described above). The database agent extracts the variables and and composes them into an RDQL query [49]:

```
SELECT
        ?r
WHERE
        (?r, <res:cuisine>, <res:BarPubBreweryCuisine>),
        (?r, <res:alcohol>, <res:FullBar>)
        (?r, <loc:city>, "Swinoujscie")
USES
  res for <http://www.agentlab.net/schemas/restaurant#>,
  loc for <http://www.agentlab.net/schemas/location#>,
  alcohol for <http://www.agentlab.net/schemas/alcohol#>
```

The query is executed against our RDF database and matching RDF triples are serialized to create an RDF/XML document:

```
:Poland_ZP_Swinoujscie_Albatros__Dyskoteka1051905696
        a res:Restaurant;
    loc:streetAddress "ul._Zeromskiego_1";
```

```
res:alcohol res:FullBar;
loc:city "Swinoujscie";
loc:country "Poland";
res:cuisine res:BarPubBreweryCuisine;
res:hours "24h in summer";
res:locationPath "Poland/ZP/Swinoujscie";
res:parsedHours "0-24|0-24|||||";
loc:phone "+48 (91) 321 18 66";
loc:state "ZP";
loc:zip "72-600";
res:title "Albatros, Dyskoteka" .
```

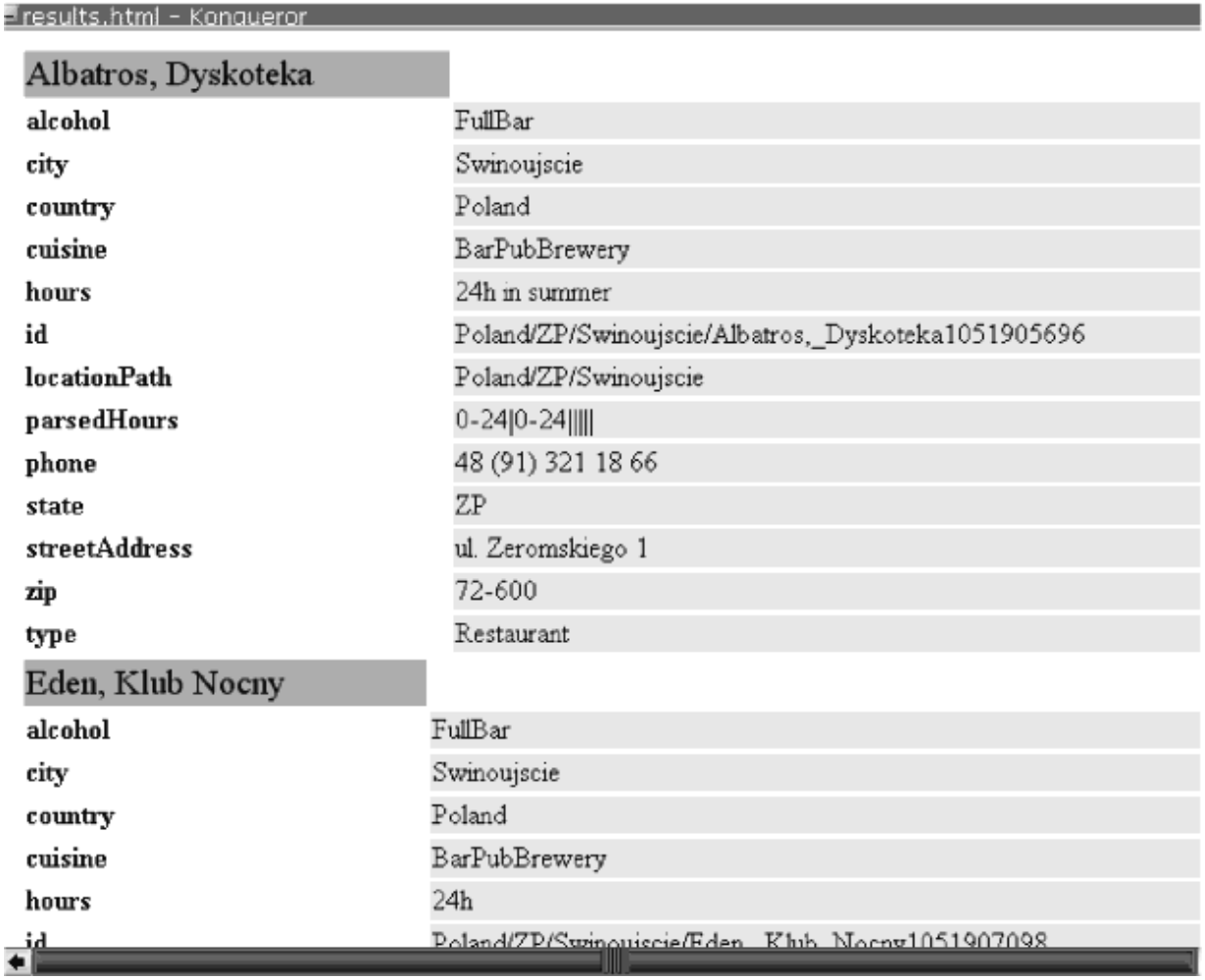


FIG. 6.3. *Results page*

The process then proceeds as outlined above. The user sees the HTML-rendered query results as a normal web page (v. Figure 6.3).

The entire chain of interactions between software agents on the server side remains transparent to the user. The problem of agent-client interaction has been successfully addressed without resorting to a proprietary agent platform or unrealistic assumptions. Furthermore, the extensive features of the general-purpose JADE agent platform allow us to exploit other properties of agent systems, such as the ability to distribute and migrate agents (specifically, the database, transformation, and personal agents) between multiple agent machines. From a conceptual perspective, the use of software agents has resulted in a highly modular and encapsulated design,

with agents for each of the tasks in the system. Furthermore, we believe that our use of the personal agent is a particularly intuitive means of representing and coordinating per-user interaction.

7. Concluding remarks. In this paper we have presented a solution to the client-agent communication problem, one of the key challenges in developing realistic agent systems that interact with user devices on the Internet. The proposed solution is based on inserting proxy agents into an HTTP server, which act as a gateway between the outside world and the agent system. Thus far we have completed the implementation for the HTML-based browser used as an I/O device in the system. In the next step we will add another class of proxy agents to process WML-based interactions as well as agent to serve Java-enabled mobile devices. We will report on our progress in subsequent papers.

REFERENCES

[1] Agentcities Network Services, http://www.agentcities.net
[2] R. Angryk, V. Galant, M. Paprzycki, M. Gordon, Travel Support System—an Agent-Based Framework, *Proceedings of the International Conference on Internet Computing IC'2002*, CSREA Press, Las Vegas, NV, 2002, pp. 719-725
[3] S. Arumugam, A. Helal, A. Nalla, *aZIMAS: Web Mobile Agent System*, http://www.harris.cise.ufl.edu/projects/publications/ma02_final.pdf
[4] Bluejade Project, http://sourceforge.net/projects/bluejade
[5] Project Bluestone; Summary, http://www.bluestone.com/downloads/pdf/06-21-01_Total-e-Server_white_paper.pdf, 2001
[6] Project Bluestone; Universal Listening Framework, http://www.hpmiddleware.com/downloads/pdf/02-27-01_ULFWhitePaper.pdf, 2001
[7] P. Buckle, FIPA and FIPA-OS Overview, Invited talk, *Joint Holonic Manufacturing Systems and FIPA Workshop*, London, September, 2000,
[8] FIPA-OS, http://fipa-os.sourceforge.net
[9] B. Burg, em Agents in the World of Active Web-services, http://www.hpl.hp.com/org/stl/maas/docs/HPL-2001-295.pdf, 2001
[10] M. Butler, F. Gianetti, R. Gimson, T. Wiley, Device Independence and the Web, *IEEE Internet Computing*, September/October, 2002, pp. 81-86
[11] ChefMoz, http://chefmoz.org/
[12] Cocoon Project, http://cocoon.apache.org/
[13] DAML Ontology Language, http://www.daml.org
[14] D. Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer-Verlag; Berlin, 2001
[15] Foundation for Intelligent Physical Agents, http://www.fipa.org
[16] FIPA, *FIPA ACL Message Structure Specification*, http://www.fipa.org/specs/fipa00061, 2001
[17] FIPA, *FIPA Agent Message Transport Envelope Representation in XML Specification*, http://www.fipa.org/specs/fipa00085
[18] FIPA, *FIPA Agent Message Transport Service Specification*, http://www.fipa.org/specs/fipa00067, 2001
[19] FIPA, *FIPA Agent Message Transport Protocol for HTTP Specification*, http://www.fipa.org/specs/fipa00084, 2001
[20] FIPA, *FIPA Agent Message Transport Protocol for IIOP Specification*, http://www.fipa.org/specs/fipa00075, 2000
[21] FIPA, *FIPA Agent Message Transport Protocol for WAP Specification*, http://www.fipa.org/specs/fipa00076, 2000
[22] FIPA, *FIPA Personal Travel Assistance Specification*, http://www.fipa.org/specs/fipa00080/XC00080B.html, 2001
[23] V. Galant, M. Gordon, M. Paprzycki, Knowledge Management in an Internet Travel Support System, *Proceedings of ECON2002*, ACTEN Press, Wejcherowo, 2002, pp. 97-104
[24] V. Galant, J. Jakubczyc, M. Paprzycki, Infrastructure for E-Commerce, *Proceedings of the 10th Conference on Extraction of Knowledge from Databases*, Wroclaw University of Economics Press, Wroclaw, Poland, 2002, pp. 32-47
[25] V. Galant, M. Paprzycki, Information Personalization in an Internet Based Travel Support System, *Proceedings of the BIS'2002 Conference*, Poznan University of Economics Press, Poznan, Poland, 2002, pp. 191-202
[26] M. Gordon, J. Jakubczyc, V. Galant, M. Paprzycki, Knowledge Management in an E-commerce System, *Proceedings of the Fifth International Conference on Electronic Commerce Research*, Montreal, Canada, October, 2002, CD, 15 pages
[27] IEEE Intelligent Systems Journal, Special Issue, Vol. 16, No. 2, 2001, http://www.computer.org/intelligent/ex2001/x2toc.htm
[28] IEEE Intelligent Systems Journal, Special Issue on Intelligent Systems for Tourism, Vol. 17, 2002, pp. 53–66
[29] M. Gordon, M. Paprzycki, A. Gilbert, Knowledge Representation in the Agent-Based Travel Support System, *Advances in Information Systems*, Springer-Verlag, Berlin, 2002, pp. 232–241
[30] Java Agent Development Environment(JADE), Telecom Lab Italia, http://jade.cselt.it and http://jade.cselt.it/papers.htm
[31] JENA, http://www.hpl.hp.com/semweb
[32] N.R. Jennings, An Agent-based Approach for Building Complex Software Systems, *CACM*, Vol. 44, No. 4, 2001, pp. 35–41
[33] LEAP Project, http://leap.crm-paris.com/
[34] P. Maes, Agents That Reduce Work and Information Overload, *CACM*, Vol. 37, No. 7, 1994, pp. 31-40
[35] Mozilla, http://www.mozilla.org
[36] D. Ndumu, J. Collins, H. Nwana, Towards Desktop Personal Travel Agents, *BT Technological Journal*, Vol. 16 No. 3, 1998, pp. 69–78

[37] H. Nwana, D. Ndumu, A Perspective on Software Agents Research, *The Knowledge Engineering Review*, Vol. 14, No. 2, 1999, pp. 1–18
[38] OASIS/ebXML Registry Services Specification v2.0, http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebrs.pdf
[39] Open GIS Consortium, Inc., http://www.opengis.org
[40] OpenGIS Catalog Services Implementation Specification, http://www.opengis.org/techno/specs/02-087r3.pdf
[41] OpenTravel Alliance, http://www.opentravel.org
[42] OWL Ontology Language, http://www.w3.org/TR/owl-ref/
[43] M. Paprzycki, R. Angryk, K. Kolodziej, I. Fiedorowicz, M. Cobb, D. Ali and S. Rahimi, Development of a Travel Support System Based on Intelligent Agent Technology, *Proceedings of the PIONIER 2001 Conference*, Technical University of Poznan Press, Poznan, Poland, 2001, pp. 243–255
[44] M. Paprzycki, C. Nistor, R. Oprea and G. Parakh, The Role of a Psychologist in E-commerce Personalization, *Proceedings of the 3rd European E-COMM-LINE 2002 Conference*, Bucharest, Romania, 2002, pp. 227-231
[45] M. Paprzycki, P. J. Kalczynski, I. Fiedorowicz, W. Abramowicz, M. Cobb, Personalized Traveler Information System, *Proceedings of the 5th International Conference Human-Computer Interaction*, Akwila Press, Gdansk, Poland, 2001, pp. 445–456
[46] S. Poslad, H. Laamanen, R. Malaka, A. Nick, P. Buckle, A. Zipf, CRUMPET: Creation of User-friendly Mobile Services Personalised for Tourism, *Proceedings of: 3G 2001—Second International Conference on 3G Mobile Communication Technologies*, 26-29 March 2001, London, UK. http://conferences.iee.org.uk/3G2001/, 2001
[47] Racoon, http://rx4rdf.liminalzone.org/Racoon
[48] RDF Primer, http://www.w3.org/TR/rdf-primer/
[49] RDQL—A Query Language for RDF, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109
[50] G. Rimassa, *Perspectives for Agent Middleware*, http://citeseer.nj.nec.com/551140.html
[51] RxPath, http://rx4rdf.liminalzone.org/RxPath
[52] RxSLT, http://rx4rdf.liminalzone.org/RxSLT
[53] W3C Semantic Web, http://www.w3.org/2001/sw/
[54] SemanticWeb Project, http://www.semanticweb.org
[55] SemanticWeb General Description, http://www.semanticweb.org/about.html#bigpicture
[56] SOAP, World Wide Web Consortium, http://www.w3.org/2002/ws, 2002
[57] J.N. Suarez, D. O'Sullivan, H. Brouchoud, P. Cros, Personal Travel Market: Real-Life Application of the FIPA Standards, *Technical Report*, BT, Project AC317, 1999
[58] J.N. Suarez, D. O'Sullivan, H. Brouchoud, P. Cros, C. Moore, C. Byrne, Experiences in the Use of FIPA Agent Technologies for the Development of a Personal Travel Application, *Proceedings of the Fourth International Conference on Autonomous Agents*, Barcelona, Spain, 2000
[59] W3C, WSDL specification, http://www.w3.org/TR/wsdl
[60] W3C, XUL specification, http://www.mozilla.org/xpfe/xptoolkit/xulintro.html
[61] W3C, XUP specification, http://www.w3.org/TR/xup/#normalop_request_ex
[62] Web Ontology (WebOnt) Working Group, http://www.w3.org/2001/sw/WebOnt/
[63] Web Ontology (WebOnt) Working Group, http://www.w3.org/2001/sw/WebOnt
[64] J. Wright, M. Gordon, M. Paprzycki, P. Harrington, S. Williams, Using the ebXML Registry Repository to Manage Information in an Internet, *Proceedings of the BIS 2003 Conference*, Poznan University of Economics Press, Poznan, Poland, 2003, 81-89

Edited by: Dan Grigoras, John P. Morrison
Received: December 17, 2002
Accepted: February 02, 2003

Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 125–139. http://www.scpe.org
ISSN 1895-1767

STATIC ANALYSIS FOR JAVA WITH ALIAS REPRESENTATION REFERENCE-SET IN HIGH-PERFORMANCE COMPUTING

JONGWOOK WOO*

Abstract. Static Analysis of aliases is needed for High-Performance Computing in Java. However, existing alias analyses regarding $*$ operator for C/C++ have difficulties in applying to Java and are even imprecise and unsafe. In this paper, we propose an alias analysis in Java that is more efficient, at least equivalent, and precise than previous analyses in C++. In the beginning, the differences between C/C++ and Java are explained and a reference-set alias representation is proposed. Second, we present *flow-sensitive* intraprocedural and context-insensitive interprocedural rules for the reference-set alias representation. Third, for the type determination, we build the type table with reference variables and all possible types of the reference variables. Fourth, a static alias analysis algorithm is proposed with a popular iterative loop method with a structural traverse of a CFG. Fifth, we show that our reference-set representation has better performance for the alias analysis algorithm than the existing *object-pair* representation. Finally, we analyze the experimental results.

Key words. Static Analysis, Alias Analysis, Java, Compiler, Parallel and Distributed Computing, Reference-Set, Flow-Sensitive, Context-Insensitive

1. Introduction. Java has become a popular language in distributed and parallel computing because it is platform independent and object-oriented. More specifically, in Java, objects are accessed by references, consequently, there might be many aliases in a piece of Java code. An alias situation occurs when an object is bound by more than two names. Thus, alias information is very useful to avoid side effects from the object bound and codes with alias information are better candidates for high performance computing such as parallelizing compiler as well as compiler optimization. Recent studies [1, 2, 3, 4, 5, 6, 8, 9, 13, 14, 15] have analyzed aliases to avoid side effects statically. With detected aliases information, we may avoid race conditions, context switch and communication overhead for parallelizing computing environment.

Previous studies [2, 3, 4, 5, 6, 8, 9, 18] proposed alias analyses for C/C++ by representing the alias relation with objects because of the concepts of pointers and pointer-to-pointer in C/C++. This research aims at improving the efficiency and the accuracy based on the safety of the alias information detected. However, the representation of alias relations based on * operator is not sufficiently optimized to apply to Java. Thus, we have proposed referred-set representation [1] that makes a more efficient and precise analysis for Java theoretically than previous methods. We can define the *precision* as the metric when all possible data are predicted statically and remove redundant data as possible. However, it might have a large time complexity in the usage of the computed alias set. Thus, in this paper, we propose an alternative alias relation, *reference-set* representation by extending our reference-pair representation which is a pair of reference variables [13]. Within a procedure, *flow-sensitive* analysis is applied. In a *flow-sensitive* intraprocedural analysis, each statement includes all the alias information at the point. The information is propagated to the next statement and subsequently computed in the CFG of a method. Among procedures, context-insensitive call graph is built. Each procedure is represented by a single node in a call graph and analyzed the node even for different calling contexts. Thus, data information is computed efficiently in context-insensitivity graph and the computed data are safe. The *context-sensitive* approach is characterized by a data flow analysis based on path-sensitivity, so each procedure may be analyzed separately for different calling contexts. Thus, we use context-insensitivity calling graph. We also can define the $safety$ as the metric when all possible data are predicted statically and collected so that any possible aliased element is not removed. At a procedure p, $May\ Aliases$ may refer to the same storage location, that is, in some execution instances of the p on some path through a flow-graph. For example, in a procedure p, when one path of p contains $x = y$ and another path $x = z$, we can say that x may refer to y or z. At a procedure p, $Must\ Aliases$ must refer to the same storage location, that is, in all execution instances of the p on all paths through a flow-graph. For example, in a procedure p, when every path of p contain only $x = y$, we can say that x must refer to y.

We analyze the existing alias representations and propagation rules in C++ [2, 3, 4, 6] in order to build much better solution in Java. Our alias analysis in Java presents three contributions for the efficiency and preciseness without losing its safety. First, we introduce the *reference-set* representation to present an alias information in Java. Second, we propose more precise data propagation rules of aliases for the *reference-set*

*Computer Information Systems Department, Simpson Tower Room 604, California State University, Los Angeles, CA 90032-8530 (jwoo5@calstatela.edu).

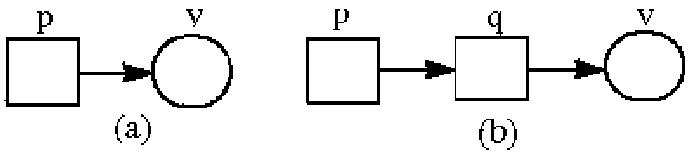


FIG. 1.1. *Relation between a pointer and an object in C++*

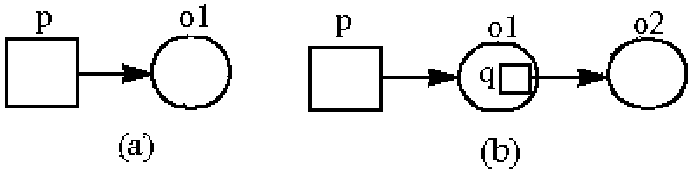


FIG. 1.2. *Relation between a reference and an object in Java*

representation. These rules are based on *flow-sensitive* and context-insensitive analysis. Finally, converting existing algorithms [1, 6, 13] to the *reference-set* representation and the rules, our calling graph (CG), control flow graphs (CFG) and a type table (TT), we propose a more precise and efficient *flow-sensitive* alias analysis algorithm.

In this paper, §2 presents the differences between pointer in C++ and reference in Java. §3 explains the existing studies for aliases. §4 introduces the *reference-set* alias representation for Java. §5 describes the structures to build our algorithm. §6 explains our propagation rules for *Flow-Sensitive* intraprocedural and Context-Insensitive interprocedural analyses. §7 shows the alias analysis algorithm and compares the complexities of our and existing algorithms. §8 shows the experimental results of the *reference-set* and existing *object-pair* representations. Finally, the conclusion is presented.

2. Pointer in C/C++ and Reference in Java. In static analysis of an object-oriented language, naming of an object has been used as data representation for data flow analysis. The object naming is considered to represent aliases in C++ and Java. In C++, static objects declare object names. Also, dynamic objects and pointer-valued objects have their own names for an alias analysis. A pointer variable name is a name to point an object that contains the address of a pointed-to object. In Fig 1.1 (a), pointer variable name p is naming a pointer-valued object that contains its address value. Dereferenced pointer p is naming the object that is pointed to by p. A variable name p that is not a pointer is naming an object that contains the address of the variable. There exist alias relations among pointer-valued objects because of pointer-to-pointer relationships. Therefore, in the previous studies [5, 6, 7], when pointer p points to an object of v, the alias relation is represented as $< *p, v >$. Fig 1.1 (b) shows that a pointer points to another pointer variable that complicates the alias analysis, where a box depicts a pointer-valued object and a circle is a non-pointer object. Those alias relations are represented as $\langle *p, q\rangle$ and $\langle *q, r\rangle$.

However, There are no pointer-to-pointer concepts and no pointer operations in Java. An object in Java is created dynamically so that the object becomes an anonymous object that does not have its own name. Thus, each object needs its own naming by binding a reference name and an object name in an alias relation. A reference is a variable that refers to an object as a pointer in C++. Fig 1.2 represents the same structure of objects and variables in Java as the Fig 1.1 in C++. In Fig 1.1, variable v can exists as an object and its object can be represented as $*v$ with the pointer operator $*$. But, in Fig 1.2 (a), the object $o1$ is referred to by the reference variable p; In Fig 1.2 (b), the object $o1$ is referred to by the variable p and the object $o2$ is referred to by the field q of the object $o1$.

Existing alias relations in C++ are similar compact [5, 6, 7] and *points-to* [2, 3, 4] representation. In this paper, we call them as *object-pair* representation because those are a pair of objects. Those relations save spaces by representing all alias relations without using an exhaustive set. Those relations can be used in Java. However, there are some problems to use those representations because only references are used to name objects in Java. Besides, existing alias representations and the analyses in C++ are based on $*$ operator. In Fig 2.1 (a), if there is an assignment statement $*x = z$, the value of the address valued object named by x is changed to

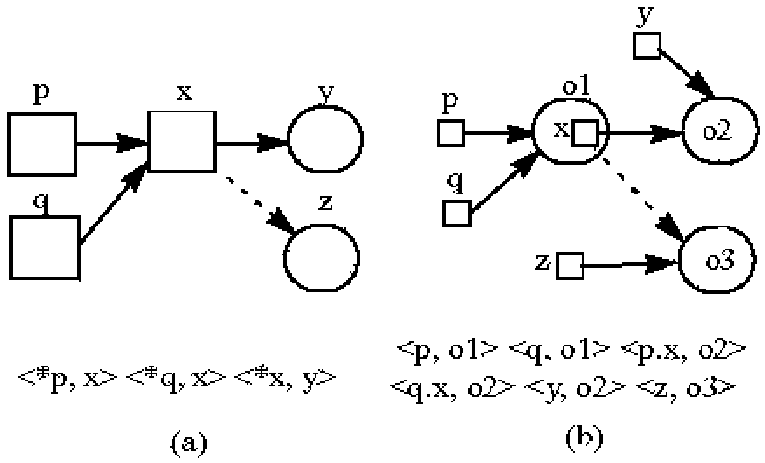


FIG. 2.1. *Difference between a pointer in C++ and a reference in Java*

the value of the address valued object bound by z. Therefore, $< *x, y >$ can be killed and a new alias relation $< x, *z >$ is generated through the compact representation rule [5, 6, 7] as follows.

*object-pair*s A_{IN} before $*x = z$:

$$A_{IN} = \{< *p, x >, < *q, x >, < *x, y >\} \tag{2.1}$$

*object-pair*s after $*x = z$ that implies $x = \&z$:

$$Kill : A_{IN}(*x) = \{\langle *x, y\rangle\} \tag{2.2}$$

$$Gen : \bigcup_{\langle *x,u\rangle \in A_{IN}, AR \in A_{IN}(*z)} \{AR(*u/*z)\} = \{\langle x, x\rangle\} \otimes \{\langle z, z\rangle\} = \{\langle *x, z\rangle\} \tag{2.3}$$

finally, *object-pair*s A_{OUT}

$$\begin{aligned} A_{OUT} &= (A_{IN} - Kill) \cup Gen = (\{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, y\rangle\} \\ &\quad -\{\langle *x, y\rangle\}) \cup \{\langle *x, z\rangle\} = \{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, z\rangle\} \end{aligned} \tag{2.4}$$

In Fig 2.1 (b), an alias relation via *compact representation* in Java is shown. If there is an assignment statement $p.x = z$ when the variable z refers to an object $o3$, we can compute alias relations with the *compact representation* rule as follows: *object-pair*s A_{IN} before $p.x = z$:

$$AIN = \{< p, o1 >, < q, o1 >, < z, o3 >, < y, o2 >, < p.x, o2 >, < q.x, o2 >\} \tag{2.5}$$

*object-pair*s after $p.x = z$:

$$\begin{aligned} &Kill : A_{IN}(p.x) = \{\langle p.x, o2\rangle\} \\ &Gen : \bigcup_{<p.x,u>\in A_{IN}, AR \in A_{IN}(z)} \{AR(u/z)\} = \{\langle p.x, p.x\rangle\} \otimes \{\langle z, z\rangle\} \\ &\qquad\qquad = \{\langle p.x, z\rangle\} \end{aligned} \tag{2.6}$$

finally, *object-pair*s A_{OUT} is

$$
\begin{aligned}
A_{OUT} &= (A_{IN} - Kill) \cup Gen \\
&= (\{\langle p, o1\rangle, \langle q, o1\rangle, \langle z, o3\rangle, \langle y, o2\rangle, \langle p.x, o2\rangle, \langle q.x, o2\rangle\} \\
&- \{\langle p.x, o2\rangle\}) \cup \{\langle p.x, z\rangle\} \\
&= (\{\langle p, o1\rangle, \langle q, o1\rangle, \langle z, o3\rangle, \langle y, o2\rangle, \langle q.x, o2\rangle, \langle p.x, z\rangle\}
\end{aligned} \tag{2.7}
$$

However, for the correct relation, $< q.x, o2 >$ of A_{OUT} should be killed. The incorrect result comes from the fact that, in Java, a reference name is used for naming an object without a dereferencing operator such as $*$ in C++. Therefore, we believe that the traditional rule is not applicable to Java to detect precise alias relations as well as *compact representation* may have more aliased elements in Java as shown in Fig 2.1 (b). To obtain the correct result in this example, $p.x$ should be recognized not only as a memory location that contains its addressed value in $< p.x, o2 >$ but also as an object that is referred to by the reference $p.x$. To solve this problem, an alias relation for an address-valued object should be presented by extending a *compact representation*. Otherwise, a data flow equation for aliases should recognize the difference. Therefore, reference names for an alias relation should be meant as dereferencing and the $< p.x, o2 >$ alias relation for the alias computation should be analyzed differently.

3. Related Works. *Pande* [2, 3] presented the first algorithm which simultaneously solved type determinations and pointer aliases with *points-to* alias set representation in C++ programs. *points-to* has the form $< loc, obj >$ where *obj* is an object and *loc* is a memory location of the object *obj*. *points-to* pair is essentially *points-to* relation as introduced by *Emami* [9]. *Emami* proposed it to reduce extraneous alias pairs generated in certain cases with alias pairs of *Landi* [18].

Carini [6] proposed a *flow-sensitive* alias analysis in C++ with *compact representation*. The *compact representation* is an alias relation that has a name object or one level of dereferencing. The *compact representation* of alias relation was introduced by *Choi* [5] to eliminate redundant alias pairs.

Chatterjee [4] presented a *flow-sensitive* alias analysis in object-oriented languages with *points-to* alias set representation in C++. It improves the efficiency and safety of *points-to* alias set representation comparing to *Pande*'s method.

The compact and *points-to* alias representation are highly similar. However, the *points-to* alias representation contains *may* or *must* alias information [5].

Woo [1] introduced a *flow-sensitive* alias analysis in Java with $referred - set$ alias representation, which is alternate to this paper. *Referred-set* is a set of objects that may be pointed by a reference variable and an alias set is a collection of *referred-set*. It is used to reduce extraneous alias pairs while applying the *compact* and *points-to* alias representations in C/C++ to Java.

4. *Reference-Set*: Alias Relation Representation. For a more precise alias analysis in object-oriented languages, the type information of the objects accessed are needed and this information can be collected more safely via alias information [2, 3]. It is known as a type inference. The type information can be used for overridden methods resolution in Java. The more precise the type information, the more precise alias analysis becomes. As shown in §2, to find a correct alias information in Java, we should present an alias representation that does not consider $*$ operator. Otherwise, we have to extend or renewal existing alias computation rules for Java. The second is not easy to implement the rules without $*$ operator. Thus, in this section, we propose the *reference-set* representation and later, we show that it improves the efficiency of the alias computation and the type inference.

DEFINITION 4.1. Reference-set *is a set of alias references that consist of more than two references which refer to an object;* $R_i = r_1, r_2, \ldots, r_j$*: for each* j*, initially* $j > 2$ *and* r_j *is a reference for an object; when* r_j *and* r_k *are in the same path and qualified expressions with a field* f*,* r_j *and* r_k *can be represented with a* $R_i.f$ *with a* reference-set R_i *for an object* i*; During data flow computing in an alias analysis,* $j > 1$ *when passing and passing back an object at a call site.*

The alias set contains the entire alias information at the statement.

DEFINITION 4.2. *Alias set is a set of* reference-set*s at a statement* s*;* $A_s = R_1, R_2, ..., R_i$

In a statement s of a program, each *reference-set* and alias set for the alias relation in Fig 4.1 are represented as follows.

$$R_1 = \{a, b\},\ R_2 = \{R_1.e, c, d\},\ R_4 = \{f.h, g\},\ A_s = \{R_1, R_2, R_4\}$$

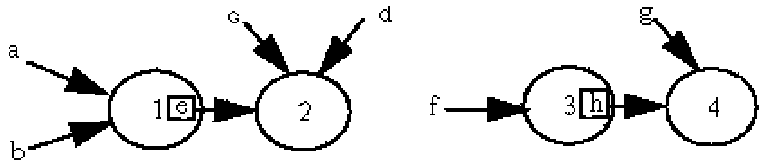


FIG. 4.1. *The relationships between reference and objects*

An alias analysis algorithm computes the alias sets in a program. Each statement collects an alias set from its predecessor and updates it with the statement itself and passes the resulting alias set to its successor(s). Since the alias computation should be iteratively done until the alias sets and a calling graph have converged for the program, it affects the efficiency of the whole algorithms. For example, there is an assignment statement $a = g$ in Fig 4.1. It means that the reference a refers to an object that the reference g refers to. Therefore, the element a of the *reference-set* R_1 is killed and the elements a should be copied to the *reference-set* R_4. The time complexity of this computation depends on the space complexity of each representation. Thus, the efficiency of whole algorithms is improved via *reference-set* representation.

5. Data-Flow Structures. Aliases can be computed with data-flow equations. For the computation of the aliases, we define our data-flow equations, calling graph (CG), and control flow graph (CFG) in this section. A *type table* contains all possible types of reference variables.

A CG is needed to compute the alias set of an interprocedural analysis between a calling and a called methods at a call statement. Our CG is a directed graph defined as $\langle N_{CG}, E_{CG}, n_{main}\rangle$, where N_{CG} is a set of nodes and each node is a method shown one time in a CG even though it may be called many times; where E_{CG} is a set of directed edges connected from caller(s) to callee(s) and one edge is connected even though a caller may invoke the callee many times and all edges are connected when many callers invoke one callee; n_{main} is the main method that executes initially in a Java program. During our algorithm proceeds, our CG grows as in the previous works [4, 5, 6, 7] by adding nodes.

CFGs can be used to compute the alias sets of the *intraprocedural* propagations. Our CFG is a directed graph defined for each method as $\langle N_{CFG}, E_{CFG}, n_{entry}, n_{exit}\rangle$, where N_{CFG} is a set of nodes with n_{entry}, n_{exit}, and each statement of the method; E_{CFG} is the set of directed edges that represent the control and alias set information between a predecessor and a successor statements; n_{entry} represents the entry node of the method; n_{exit} represents the exit node of the method.

In our CFG, seven node types are proposed based on their purposes: *Entry* that is the n_{entry} of the CFG, *Exit* that is the n_{exit} node, *Assignment statement, Call statement, Return Statement, Flow construct node (if, while etc.)*, and *Merging nodes*.

The *flow construct node* is a node which signifies the start of the if or *while* clause. In an If node, each clause is branched from the node. All the branched clauses are merged into a merging node. In a *while* node, a *merging node* is not necessary and a directed edge is connected from the last node of the while loop to the *flow construct node*.

Reference variables dynamically refer to objects in Java. Thus, the types of a reference variable are determined statically at the type declaration and dynamically during the processing of the algorithm. A *type table* is built during the process and it contains three columns: the reference variable, its declared type, and its overridden method types. Type inference can be processed with a *type table* which contains the declared and dynamic types of each reference variable. The declared type represents static and shadowed variable type information of a reference variable. The dynamic types represent possible overridden method types of the reference variable. Types of each reference variable can be given in a constant time.

6. *Flow-Sensitive* Context-Insensitive Rules for *Reference-Set*. The propagation and computation of alias information is made through the nodes in a CFG of each method. Let $in(n)$ and $out(n)$ be the input alias set of a node n transferred from predecessor nodes and output alias set held on *exit* from a node n respectively.

$$in(n) = \bigcup out(pred(n))$$
$$out(n) = Trans(in(n)) = Mod_{gen}[Mod_{kill}(in(n))] \tag{6.1}$$

In this equation, $pred(n)$ represents a predecessor node of the node n. Mod_{kill} denotes the alias set modified after killing some *reference-set*s of $in(n)$ and Mod_{gen} is the subsequent alias set after generating the new *reference-set*s on Mod_{kill}.

6.1. *Flow-Sensitive* Intraprocedural Analysis. The propagation rules for *intraprocedural* analysis are described below for every CFG node type except an *entry* and a *call statement node* type. The rule consists of premises and conclusions divided by a horizontal line. The premise can have the form of conditional implication that is interpreted as follows: when a given condition holds, the implied equation has a meaning and can be solved. When all premises hold, the equations in the conclusions are solved for $out(n)$.

First, we define a *flow construct* and *merging node* type rule in (6.2): n_{pred} is a predecessor set of node n. Given n_{pred}, $out(n)$ of node n is the union of all predecessor node sets.

$$in(n) = \bigcup_{p \in n_{pred}} out(p)$$

$$\frac{n_{pred} : predecessor\ node\ of\ n}{out(n) = in(n)} \tag{6.2}$$

The next rule (6.3) concerns the node type of an *assignment statement*.

$$\begin{gathered} in(n) = out(npred) \\ n_{pred} : predecessor\ node\ of\ n \\ x = LHS, \\ y = RHS, \\ \forall\, i, j\ Ri, Rj \in in(n) \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\, x \in R_i\} \\ \cup \{R_i \mid\ kill\, R_j.f \in R_i\ when\, q \in R_j\ and\, x = q.f\}] \text{.....①} \\ \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \wedge [KILL(in(n)) = \{x,\ R_j.f\}], \text{.....②} \\ \forall k\ R_k \in in(n) \rightarrow [Modgen(in(n)) = \{R_k \mid R_k = R_k \cup KILL(in(n))\ when\, y \in R_k\}] \\ \wedge [in(n) = in(n) - Mod_{gen}(in(n))], \text{.....③} \\ \hline out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) \end{gathered} \tag{6.3}$$

LHS and RHS respectively stand for the left hand side and the right hand side of an assignment statement. $KILL(in(n))$ is a *reference-set* of references killed by $Mod_{kill}(in(n))$. At a statement ①, $\{R_i \mid kill\ x \in R_i\}$ is to remove the element x from the set R_i. $out(n)$ of the node n is a union of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and $in(n)$.

In order to show how the above rule can be applied to alias analysis, we analyze an *assignment statement* $a.e = f.h$ in a statement of Fig 4.1. Initially, *reference-set* R_1, R_2, R_3 and alias set $in(n)$ are expressed as follows for the statement:

$$\begin{gathered} R1 = \{a, b\}\ \ R2 = \{R_1.e, c, d\}\ \ R3 = \{f.h, g\} \\ in(n) = \{R_1, R_2, R_3\} \end{gathered}$$

Because LHS is a qualified expression related to both R_1 and R_2, $Mod_{kill}(in(n))$, $in(n)$, and $KILL(in(n))$ are computed based on ① and ② as follows:

$$\begin{gathered} R_1 = \{a, b\}\ and\ R_2 = \{R_1.e, c, d\}\ then\ R_2 = \{c, d\} \\ Mod_{kill}(in(n)) = \{R_2\},\ in(n) = \{R_1, R_3\},\ KILL(in(n)) = \{R_1.e\} \end{gathered}$$

Since R_3 includes RHS, $Mod_{gen}(in(n))$ and $in(n)$ are computed based on ③ as follows:

$$Mod_{gen}(in(n)) = \{R_3 \mid R_3 = R_3 \cup \{R_1.e\} = \{R_1.e, f.h, g\}\} = \{R_3\},\ in(n) = \{R_1\}$$

Finally, $out(n)$ is the union set of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and $in(n)$ as follows:

$$out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) = \{R_2, R_3, R_1\}$$
$$when\ R_1 = \{a, b\},\ R_2 = \{c, d\},\ R_3 = \{R_1.e, f.h, g\}$$

The following is the another example to detect *reference-set* of Fig 2.1 (b) by assuming the current node as n with the assignment statement $p.x = z$:

$$R_1 = o1 = \{p, q\},\ R_2 = o2 = \{p.x, q.x, y\} = \{R_1.x, y\},\ R_3 = o3 = \{z\}$$
$$in(n) = \{R_1, R_2, R_3\}$$

Because LHS is a qualified expression related to R_1, $Mod_{kill}(in(n))$, $in(n)$, and $KILL(in(n))$ are computed based on ① and ② of *Assignment Node* rule (6.3) as follows:

$$R_1 = \{p, q\}\ and\ R_2 = \{R_1.x, y\}\ then\ R_2 = \{y\}$$
$$Mod_{kill}(in(n)) = \{R_2\}\ in(n) = \{R_1\}\ KILL(\ in(n)) = \{R_1.x\}$$

Since R_3 includes RHS, $Mod_{gen(in(n))}$ and $in(n)$ are computed based on ③ of *Assignment Node* rule (6.3) as follows:

$$Mod_{gen}(in(n)) = \{R_3 \mid R_3 = R_3 \cup \{R_1.x\}\} = \{z, R_1.x\} = R_3,\ in(n) = \{R_1\}$$

Thus, $out(n)$ is the union set of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and $in(n)$ as follows:

$$out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) = \{R_2, R_3, R_1\}$$
$$when\ R_1 = \{p, q\},\ R_2 = \{y\},\ R_3 = \{z, R_1.x\}$$

Finally, $out(n)$ has the following $referenceset$, which has the correct aliased elements:

$$out(n) = \{R_1, R_2, R_3\}\ when\ R_1 = \{p, q\},\ R_2 = \{y\}, R_3 = \{z, R_1.x\}$$

With this example for Fig 2.1 (b), we have shown that our reference set of $out(n)$ is more precise than the aliased elements A_{OUT} of the *compact representation* shown in the §2.

The rule (6.4) for the *return statement node* type is presented as follows with the *reference-set* of a return variable r. In the rule, $LOCAL$ stands for a local variable set defined in a method M such as local variable and formal parameter variables.

$$\frac{\begin{array}{c} in(n) = out(npred) \\ n_{pred} : predecessor\ node\ of\ n \\ M : callee,\ LOCAL(M) = \{v \mid v\ is\ a\ local\ variable\ of\ M\} \\ \forall i\ R_i \in in(n)\ for\ r :\ return\ reference \\ \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\ for\ x \in LOCAL(M)\}] \wedge \\ [R_r = \{R_r | kill\ x \in R_r\ for\ x \in LOCAL(M)\ when\ r \in R_r\}], \end{array}}{out(n) = Mod_{kill}(in(n))} \tag{6.4}$$

The next is the rule (6.5) for an *exit node* type.

$$in(n) = \bigcup_{p \in n_{pred}} out(p),$$
$$n_{pred} :\ predecessor\ node\ of\ n,$$

$$\frac{\begin{array}{c} M : callee,\ LOCAL(M) = \{v \mid v\ is\ a\ local\ variable\ of\ M\}, \\ \forall i\ R_i \in in(n) \\ \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in Ri\ for\ x \in LOCAL(M)\}] \\ \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \end{array}}{out(n) = Mod_{kill}(in(n)) \cup in(n)} \tag{6.5}$$

6.2. Context-Insensitive Interprocedural Analysis. Interprocedural propagation rules should be considered for a *call statement node* and an *entry node*. The data flow of an alias set in a *call statement* denotes that an alias information of the statement is propagated to a called method and it affects an alias information of the called method. The affected information are passed back to the *call statement* of the calling method after computing the alias set of the called method. The alias set from the called method modifies the alias set of the *call statement* when the return alias set includes non-local variables and actual parameters.

We virtually divide a call node into a *precall node* and a *postcall node* to simplify the computation of a call statement. A *precall node* collects an alias set from a *predecessor node* of a current call node and computes its own alias set $out(n)$ with the collected set. This alias set is propagated to the *entry node* of the called method. During the propagation, the $reference - sets$ for references which are inaccessible from the called method are killed. Since this set is an input of the *postcall* node and is not modified, it does not need to propagate to the called method. The $out(n)$ of the *precall* node is not propagated to the *postcall* node because the called method might modify the set. As in previous approaches [2, 3, 4], if we do not kill the alias relations affected by the called method for the subsequent analysis, it might build nonexistent call relations and cause the subsequent analysis to become inefficient.

A *postcall* node collects the modified kill set of the *precall* node and exit nodes alias set of all possible called methods. The following rule (6.6) computes an out set of a *precall* node.

$$\frac{\begin{array}{c} in(n) = out(n_{pred}), \\ n_{pred} : predecessor\ node\ of\ n \\ RHS = E_c.M_c, \\ RHS = M_c, \\ \forall i,\ a_i = the\ ith\ actual\ parameter\ of\ the\ callee\ Mc, \\ \forall i,\ f_i = the\ ith\ formal\ parameter\ of\ the\ callee\ M_c, \\ \forall i,\ R(a_i) \in in(n) \rightarrow [R_{pass}(a_i) = \{a_i, f_i\}] \wedge [R(a_i) = R(a_i) - R_{pass}(a_i)], \\ RHS = M_c,\ \forall i,\ R(a_i) \in in(n),\ v\ is\ a\ non\ local\ variable\ in\ the\ callee, \\ M_c \rightarrow [R(v) = R(v) - \{v\}] \wedge [R_{pass}(v) = \{v\}] \wedge [PASS(M_c) = \cup\{R_{pass}(ai), R_{pass}(v)\}], \\ RHS = E_c.M_c, \forall i,\ R(a_i) \in in(n) \forall f, \\ R(E_c.f) \in in(n) \rightarrow [R(E_c.f) = R(E_c.f) - \{E_c.f\}] \wedge \\ [R_{pass}(Ec.f) = \{Ec.f\}] \wedge [PASS(M_c) = \cup\{R_{pass}(a_i), R_{pass}(E_c.f)\}] \end{array}}{out(n) = in(n)} \tag{6.6}$$

$PASS(M_c)$ represents the set of actual, formal parameters and non-local variables in a called method M_c.$R_{pass}(a_i)$ is a set of reference variables accessible by a called method when passing from a caller to the called method M_c.$R_{pass}(v)$ is a set of non-local variables accessible by a called method in the called method M_c.

In the following propagation rule (6.7) of an *entry node*, the propagated set can be computed as in an *assignment statement node*. $PRECALL(M_c)$ is a *precall node* of *call statement node*s that invoke this called method node. This set can be computed by considering ingoing edges of the called method M_c in a CG. An entry node merges alias sets from the *precall* nodes and then propagates the merged set to its subsequent node.

$$\frac{\begin{array}{c} in(n) = \bigcup_{p \in PRECALL(M_c)} PASS(p), \\ PRECALL(M_c) : a\ precall\ node\ of\ the\ callee\ M_c \end{array}}{out(n) = in(n)} \tag{6.7}$$

The rule (6.8) of the *postcall node* is defined as follows.

$$
\begin{gathered}
in(n) = \bigcup_{p \in n_{precall}} out(n_{precall}) \\
n_{precall} : \ a\ precall\ node\ of\ n \\
RHS = E_c.M_c \\
\rightarrow FIELD(E_c) = \{f \mid \ f\ is\ a\ field\ name\ in\ an\ object\ referred\ by\ E_c\}, \\
RHS = M_c \rightarrow FIELD(E_c) = \phi, \\
RHS = new\ M_c \rightarrow FIELD(E_c) = \phi \wedge A(r), \\
EXIT(M_c) = \{e \mid e\ is\ an\ exit\ alias\ set\ from\ a\ possible\ callee\ method\ M_c\}, \\
LHS = \phi, \forall R_{passb} \in EXIT(M_c) \\
\rightarrow [R_{passb} = R_{passb} - \{v \mid v\ is\ a\ local\ variable\ in\ the\ callee\ M_c\}] \\
\wedge [EXIT(M_c) = \bigcup_{for\ all\ M_c} R_{passb}], \\
LHS \neq \phi, \ \forall R_{passb} \in out(return\ node) \\
\rightarrow [R_{passb} = R_{passb} - \{v \mid v\ is\ a\ local\ variable\ in\ the\ callee\ M_c\}] \\
\wedge [EXIT(M_c) = \bigcup_{for\ all\ M_c} R_{passb}], \\
\forall i\ R_i \in EXIT(M_c), \forall j R_j \in in(n) \rightarrow [R_i \mid R_i = R_i \cup R_j\ when\ i = j] \\
\wedge [EXIT(M_c) = EXIT(M_c) - R_i] \wedge [in(n) = in(n) - R_j], \\
exit(RHS) = \bigcup_{e \in EXIT(Mc)} out(e) \cup \bigcup_{p \in n_{precall}} out(precall\ node) \cup \bigcup_{for\ all\ i} R_i, \\
LHS = \phi \rightarrow out = exit(RHS), \\
LHS = x, \forall i, j\ R_i, R_j \in exit(RHS), R(RHS) \in exit(RHS) \\
\rightarrow [Mod_{kill}(exit(RHS)) = \{R_i \mid kill\ x \in Ri\} \\
\cup \{R_i \mid kill\ R_j.f \in R_i\ when\ q \in R_j\ and\ x = q.f\}] \wedge [KILL(exit(RHS)) = \{x, R_j.f\}] \\
\wedge [exit(RHS) = exit(RHS) - Mod_{kill}(exit(RHS))], \\
R(RHS) \in exit(RHS) \rightarrow [Mod_{gen}(exit(RHS)) \\
= \{R(RHS) \mid R(RHS) = R(RHS) \cup KILL(exit(RHS))] \\
\wedge [exit(RHS) = exit(RHS) - Mod_{gen}(exit(RHS))] \\
\underline{\wedge [out = Mod_{kill}(exit(RHS)) \cup Mod_{gen}(exit(RHS)) \cup exit(RHS)]} \\
out(n) = out
\end{gathered}
\tag{6.8}
$$

$Exit(RHS)$ is a set of exit nodes of all possible called methods as explained before. We can compute $exit(RHS)$ in a CG by integrating all $out(precall\ node)$ and outgoing edges from callers and their exit nodes. Out is an alias set of the exit node of a callee. If we assume the Fig 6.1 (a) as a status after executing a statement s, the alias set A_s of the statement s is:

$$
\begin{gathered}
A_s = R_2, R_3 \\
where\ R_2 = \{a.f, b, c, R_3.f\}\ and\ R_3 = \{R_2.f, c\}
\end{gathered}
$$

After executing the call statement t in Fig 6.1 (b), the result alias set of its *precall node* can be computed in the following sequence of rule applications:

$$
\begin{gathered}
in(t) = \{R_2, R_3\}, \\
RHS = a.update(c), \\
a_i = c, f_i = i, \\
R_{pass}(ai) = \{c, i\}, R(a_i) = R_2 = R_2 - \{c\} = \{a.f, b, R_3.f\}
\end{gathered}
$$

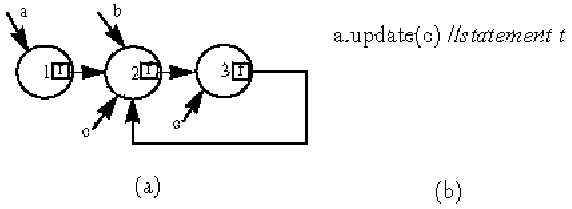


FIG. 6.1. *Example of an interprocedural analysis*

$$
\begin{gathered}
or\ R(a_i) = R_3 = R_3 - \{c\} = \{R_2.f\}, \\
R(a.f) = R_2 = R_2 - \{a.f\} = \{b, R_3.f\}\ and\ R_{pass}(a.f) = a.f, \\
PASS(a.update) = \{R_{pass}(ai), R_{pass}(a.f)\}, \\
out(t) = \{R_2, R_3\}
\end{gathered}
$$

The $PASS(a.update)$ of the precall node propagates to the *entry node* of the callee *update*(). The result alias set of the *exit node* can be computed as follows:

$$
\begin{gathered}
R_{pass}(a_i) = \{c, i\}, R_{pass}(a.f) = \{a.f\}, \\
R_{pass}(a_i) = R_{pass}(a.f) = \{c, i, a.f\} = R_{pass}(R_2)\ for\ R_2, \\
R_{pass}(ai) = \{c, i\} = R_{pass}(R_3)\ for\ R_3, \\
in(u) = \{R_{pass}(R_2), R_{pass}(R_3)\}, \\
R(b) = \{b, i.f, c.f\}, \\
out(u) = updateexit = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\}
\end{gathered}
$$

The result set of the *postcall node* at the statement t is computed with the exit alias set of the *update*() and the propagation rule of the postcall node as follows:

$$
\begin{gathered}
in(t) = out(t_{precall}) = \{R_2, R_3\}\ where\ R_2 = \{b, c.f\}, R_3 = \{c.f\}, \\
FIELD(a) = \{a.f\}, \\
EXIT(update) = update_{exit} = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\}, \\
where\ R_{pass}(R_2) = \{c, i, a.f\}\ and\ R_{pass}(R_3) = \{c, i\}, \\
R(b) = R_{passb} = \{b, c.f\},\ R_{passb}(R_2) = \{c, a.f\},\ R_{passb}(R_3) = \{c\}
\end{gathered}
$$

for the caller,

$$
\begin{gathered}
EXIT(update) = \{R(b), R_{passb}(R_2), R_{passb}(R_3)\}, \\
R_2 = R_2 \cup R_{passb} \cup R_{passb}(R_2) = \{b, R_3.f, c.f, c, a.f\}, \\
R_3 = R_3 \cup R_{passb} \cup R_{passb}(R_3) = \{R_2.f, b, c.f, c\},
\end{gathered}
$$

Thus, $out(t) = R_2, R_3$

7. Static Analysis Algorithm for Aliases. Algorithm 7.1 represents our alias analysis algorithm. This algorithm is adapted on the interprocedural type analysis algorithm [6]. It is one of iterative methods for interprocedural data flow analysis based on a *CG* [10]. The iterative algorithm executes its computation by visiting all nodes of a *CG* in order until fixed point of the data status and nodes are achieved.

Our algorithm traverses each node of a *CG* in a topological order and a reverse topological order in order to possibly shorten the execution time for the fixed point [5, 6, 7]. The ending point in our algorithm means that the topology of a *CG* and alias set $out(n)$ are not changed anymore.

The set $TYPES$ represents the possible class types for a callee to build a safe *CG*. $TYPES_{table(r)}$ is a set of dynamic types of a reference variable r in a *type table*. The reference r also maintains its static type in the

Algorithm 7.1 StaticAliasAnalysis

```
construct an initial CG with main method;
repeat
  for all node n ∈ N_cfg(T.M) in structural order do
    for all node n ∈ N_cfg(T.M) in structural order do
      if n is a call statement node then
        if (RHS = E_c.M_c) then
          compute the set of inferred types from the reference-set for E_c.
          compute the set TYPES resolved from the inferred types and class hierarchy.
        else if (RHS = M_c) then
          TYPES := {T}
        else if (RHS = newM_c) then
          TYPES := {M_c}
        end if
        if LHS exists then
          {TYPES_table(LHS) = TYPES_table(RHS);}
        end if
        for all type t ∈ TYPES do
          if t.M_c is not in CG then
            create a CG node for Mc;
          end if
          if no edge from T.M to t.M_c with a label n then
            connect an edge from T.M to t.Mc with a label n;
          end if
        end for
        compute out(n_precall) for a precall node n_precall;
        compute out(n_postcall) for a postcall node n_postcall;
      else
        if n is an assignment statement node then
          TYPES_table(LHS) = TYPES_table(RHS);
        end if
        if n is a merging statement node then
          TYPES_table(LHS) = TYPES_table(LHS) + TYPES_table(RHS);
        end if
        compute out(n) using data-flow equation and propagation rule;
      end if
    end for
  end for
until CG and alias set for every CFG node converge
```

type table. Inner loop is for *intraprocedural* alias analysis for each method. While proceeding the inner loop, each node of the CFG of a method is traversed with computing an alias set of each node and the computed alias set is propagated to the next node. Each node in our algorithm is visited on structural order for this. Structural order is defined that while visiting nodes from an entry node to an exit node, for the if flow construct node, each branch is traversed first then finally its merging node is visited. We improve the efficiency with the structural order than the previous work [6].

When the method of a *call statement node* is an overridden method, its resolution should be considered for the safety of the alias set. It is not possible to precisely predict the dynamic overridden method statically. However, we can store all possible types of each reference into a *type table* during computation. Thus, we can safely predict all possible methods invoked with the *type table*. If the method is defined in a type inferred by this type inference, the method defined is a resolved method. By adding all possible methods to the CG via the searching, we can update the CG.

An alias set of each node can be computed as a result alias set $out(n)$ with type information and our

TABLE 8.1
Characteristics of hosts

	Kottos	Ceng	Asadal
Host Type	RS6000	Sun4	Windows 2000
OS	AIX4	SunOS5.6	Windows NT
Java VM	JDK-1.1.1	JDK-1.2.1 _02	JDK-1.2.2

data-flow equations of the propagation rules. Our algorithm has three outer loops. For the most outer loop, R_n and A_r are the number of $reference-sets$ and the maximum number of aliased reference variables for each *reference-set*. We can estimate the worst time complexity of the loop as $O(Rn \times Ar \times Ecg)$ - E_{cg} is the number of edges in a CG. For the second outer loop, the time complexity becomes $O(N_{cg})$ if N_{cg} is the final number of nodes in a CG. For the most inner loop, the time complexity is $O(N_{cfg})$ if N_{cfg} is the maximum number of nodes in a CFG that consists of the maximum number of nodes.

The time complexity of a set of inferred types is $O(R_m)$ when R_m is the number of reference variables in a program code. The time complexity for the possible method resolution is $O(T_i \times H)$ when T_i is the maximum number of subclasses for a superclass and H is the maximum number of the levels in its hierarchy. The time complexity for the resolution of overridden methods and the updating of a CG is $O(T_i \times (H + Ncg + Cc))$ when C_c is the maximum number of call statements to invoke same called methods in a calling method. The worst time complexity of a *precall* and a *postcall* nodes is $O(R_p \times R)$ when R_p is the maximum number of *reference-set*s propagated and R is the maximum number of reference variables in R_p on a call statement.

Therefore, the worst time complexity of the main algorithm is $O(R_n \times A_r \times E_{cg} \times N_{cg} \times N_{cfg} \times (R_p \times R \times R_m + T_i \times (H + N_{cg} + C_c)))$.

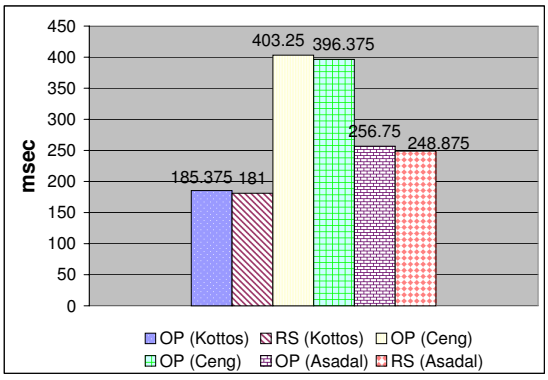


FIG. 8.1. *Run Time of Overridden Methods*

8. Experimental Results. We have executed benchmark codes on alias analysis algorithms with the *reference-set* and the existing *object-pair* representations [2, 3, 4, 6, 7]. We only focus on time for the experiment because the alias detected for these two executions are implicitly the same—the benchmark does not have the indirect object relations that may generate imprecise aliases with the existing work. The algorithm is the part of a Java compiler. Thus, the execution time is to measure the compilation time of the codes. We believe that the theoretic approach of this paper is enough to show our *reference-set* representation is both safe and precise without losing the efficiency. However, we have the three benchmark codes executed, which are: $Overridden\ Methods$, $Binary\ Tree$, and $Ray\ Tracer$. It is to present that our approach runs at least with the similar efficiency comparing to the existing *object-pair* representations [2, 3, 4, 6, 7]. Each benchmark code is executed on three different hosts $Kottos$, $Ceng$, and $Asadal$, which is to collect many experimental results. Table 8.1 presents properties of the hosts for those benchmark codes. $Kottos$ is RS6000 IBM machine; $Ceng$

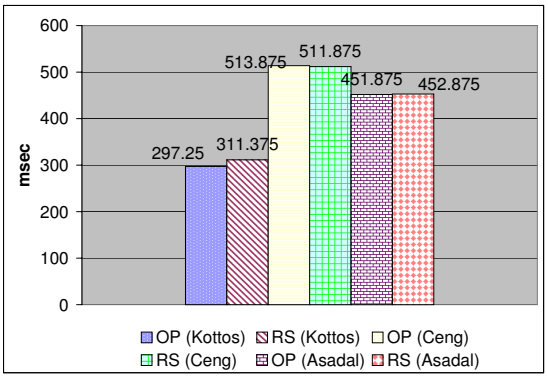


FIG. 8.2. *Run Time of Ray Tracer*

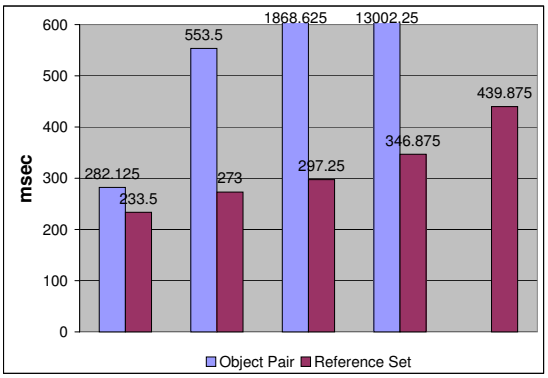


FIG. 8.3. *Run Time of Binary Tree at Kottos*

is Sun4 Unix machine; *Asadal* is Windows machine. The alias analysis systems are built on JavaCC (Java Compiler Compiler) [11] and JTB (Java Tree Builder) [12]. JavaCC is the parser generator and JTB is a syntax tree builder to be used with the JavaCC parser generator. It automatically generates a JavaCC grammar with the proper annotations to build the syntax tree during parsing [12]. The syntax tree is extended by adding the data structures of *reference-set* and *object-pair* representations with class structures of TT and CFG [13].

Overridden Methods is written in C++ initially by *Carini* [6] and adapted in Java by ourselves. It has conditional statements and overridden methods. *Binary Tree* is provided by Proactive group [14]. The binary tree contains many conditional statements and recursive calls that generate potential aliases dynamically. Both can be used to measure the safety, preciseness, and efficiency of the algorithms. *RayTracer* is one of Java Grande's benchmarks to measure the performance of a 3D raytracer [15].

Fig 8.1 presents the execution times of *Overridden Methods*. For all hosts, the execution time of *reference-set* is faster than *object-pair* because our *Type Table* has more efficient structure to search possible types of methods than Carini's [6]. Fig 8.2 is the execution times of *Ray Tracer*. It implies that the benchmark codes such as *Ray Tracer*, which do not contain many aliased references among objects and which is for JVM performance measurement, do not have any big difference in the execution time of alias analysis for any alias representation. For *Binary Tree*, Fig 8.3, Fig 8.4, and Fig 8.5 show that the execution time of *reference-set* becomes faster than *object-pair* relatively as the depth of the recursive calls increase. For *Kottos* of Fig 8.3,

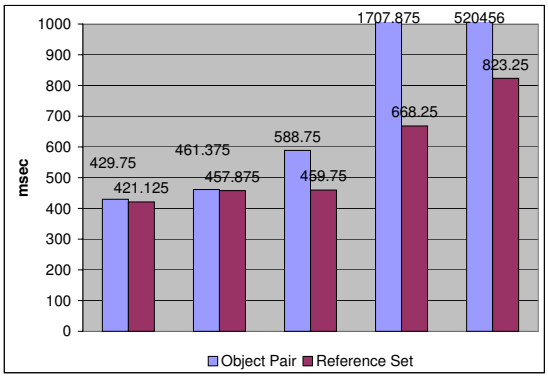


FIG. 8.4. *Run Time of Binary Tree at Ceng*

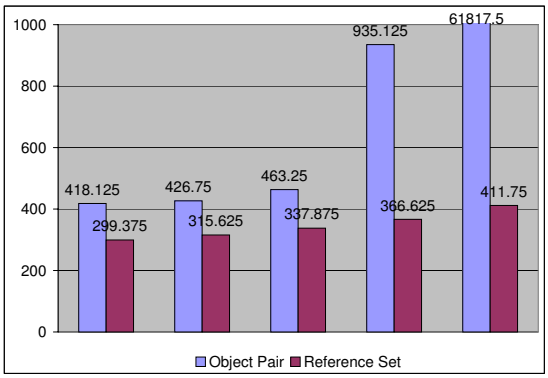


FIG. 8.5. *Run Time of Binary Tree at Asadal*

object-pair is not measurable because the execution time is too long. The result of *Binary Tree* for each host is reasonable because the performance test shows that Windows and Sun hosts are much faster than IBM for Java benchmark [19, 20]. As a result, we can see that *reference-set* is more efficient than *object-pair* for benchmark which has many aliased objects such as *Binary Tree*. Besides, it does not negatively affect the performance for benchmark which does not have many aliased objects such as *Overridden Methods* and *Ray Tracer.*

9. Conclusion. We propose the *flow-sensitive context-insensitive* static analysis algorithm for aliases with *reference-set* alias representation in Java by adapting existing alias analyses [6, 7] in C or C++. We show theoretically that the algorithm is more precise, safe, and efficient than previous studies [2, 3, 4, 6, 7] by using the *reference-set* alias representation, the structural traverse of a *CFG*, and the data propagation rules for the representation. Finally, we show in the experimental results that our approach is more efficient, at least equivalent, comparing to the previous studies [2, 3, 4, 6, 7].

REFERENCES

[1] JEHAK WOO, JONGWOOK WOO AND JEAN-LUC GAUDIOT, *Flow-Sensitive Alias Analysis with Referred-Set Representation for Java*, The Fourth International Conference/Exhibition on High Performance Computing in Asia, pp 485-494, May 2000.
[2] H. D. PANDE AND B. G. RYDER, *Static Type Determination and Aliasing for C++*, LCSR-TR-236, Rutgers Univ., 1995.

[3] H. D. Pande and B. G. Ryder, *Data-flow-based Virtual Function Resolution*, Static Analysis: Third International Symposium (SAS'96), LNCS 1145, Sept., 1996.
[4] R. Chatterjee and B. G. Ryder, *Scalable, flow-sensitive type inference for statically typed object-oriented languages*, Technical Report DCS-TR-326, Rutgers Univ., Aug. 1997.
[5] Jong-Deok Choi, Michael Burke, and Paul Carini, *Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects*, The 20th ACM SIGACT-SIGPLAN Symposium on POPL, pp 232-245, January 1993.
[6] Paul Carini and Harini Srinivasan, *Flow-Sensitive Type Analysis for C++*, Research Report RC 20267, IBM T. J. Watson Research Center, November 1995.
[7] Michael Burke, Paul Carini, and Jong-Deok Choi, *Interprocedural Pointer Alias Analysis*, Research Report RC 21055, IBM T. J. Watson Research Center, December 1997.
[8] Barry K. Rosen, *Data flow analysis for procedural languages*, JACM, 26(2):322-344, April 1979.
[9] M. Emami, R. Ghiya, and L. J. Hendren, *Context-sensitive interprocedural point-to analysis in the presence of function pointers*, SIGPLAN '94 Conference on Programming Language Design and Implementation, pp 242-256, 29(6), 1994
[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Academic Press, July 1997.
[11] Sun Micro Systems, *JavaCC, The parser Generator*, http://www.suntest.com/JavaCC/, V0.8pre2, 1998.
[12] Purdue University, *West Lafayette, Indiana, USA, Java Tree Builder*, http://www.cs.purdue.edu/jtb/index.html, 2000.
[13] Jongwook Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, *Alias Analysis On Type Inference For Class Hierarchy In Java*, The 24th ACSC 2001, pp 206-214, Jan 29-Feb 2, 2001.
[14] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, *Alias Analysis For Java with Reference-Set Representation*, The 8th ICPADS 2001, pp 459-466, June, 2001.
[15] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, *Alias Analysis For Exceptions In Java*, The 25th ACSC 2002, pp 321-330, Jan, 2002.
[16] Inria, *ProActive*, Sophia, http://www.inria.fr/oasis/ProActive
[17] Java Grande Forum, http://www.javagrande.org/
[18] W. Landi and B. G. Ryder, and S. Zhang, *A Safe Approximating Algorithm for interprocedural Pointer Aliasing*, in Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pp 235-248, June 1992.
[19] Pendragon Software *CaffeineMark 3.0*, http://www.pendragon-software.com/pendragon/cm3/results.html
[20] National Institute Standards and Technology, *SciMark 2.0*,http://math.nist.gov/cgi-bin/ScimarkSummary

Edited by: Andrzej Goscinski
Received: December 17, 2002
Accepted: February 02, 2004

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2_ε using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the PDCP WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.