CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2011; **23**:17–37
Published online 23 April 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1592

JEL: unified resource tracking for parallel and distributed applications

Niels Drost*, †, Rob V. van Nieuwpoort, Jason Maassen, Frank Seinstra and Henri E. Bal

Department of Computer Science, VU University, Amsterdam, The Netherlands

SUMMARY

When parallel applications are run in large-scale distributed environments, such as grids, peer-to-peer (P2P) systems, and clouds, the set of resources used can change dynamically as machines crash, reservations end, and new resources become available. It is vital for applications to respond to these changes. Therefore, it is necessary to keep track of the available resources—a problem which is known to be notoriously difficult. In this article we argue that resource tracking must be provided as the standard functionality in the lower parts of the software stack. We propose a general solution to resource tracking: the Join–Elect–Leave (JEL) model. JEL provides *unified* resource tracking for parallel and distributed applications across environments. JEL is a simple yet powerful model based on notifying when resources have *Joined* or *Left* the computation. We demonstrate that JEL is suitable for resource tracking in a wide variety of programming models, ranging from the fixed resource sets traditionally used in MPI-1 to flexible grid-oriented programming models. We compare several JEL implementations, and show these to perform and scale well in several real-world scenarios involving grids, clouds and P2P systems applied concurrently, and wide-area systems with failing resources. Using JEL, we have won the first prize in a number of international distributed computing competitions. Copyright © 2010 John Wiley & Sons, Ltd.

Received 10 March 2009; Revised 2 March 2010; Accepted 6 March 2010

KEY WORDS: resource tracking; programming models; parallel applications

1. INTRODUCTION

Traditionally, supercomputers and clusters are the main computing environments[‡] for running high performance parallel applications. When a job is scheduled and started, it is assigned a number of machines, which it uses until the computation is finished. Thus, the set of resources used for an application in these environments is generally fixed.

In the recent years, parallel applications are also run on large-scale grid systems [1], where a single parallel application may use resources across multiple grid sites simultaneously. Recently, peer-to-peer (P2P) systems [2], desktop grids [3], and clouds [4] are also used for running parallel and distributed applications. In all such environments, resources may become unavailable at any time, for instance when machines fail or reservations end. In addition, new resources may become

*Correspondence to: Niels Drost, Department of Computer Science, VU University, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands.
†E-mail: niels@cs.vu.nl

‡We will use the term *environment* for collections of compute resources, such as supercomputers, clusters, grids, desktop grids, clouds, and P2P systems, throughout this article.

Contract/grant sponsor: Dutch Ministry of Education, Culture and Science (OC&W)
Contract/grant sponsor: Netherlands Organization for Scientific Research (NWO); contract/grant number: 612.060.214

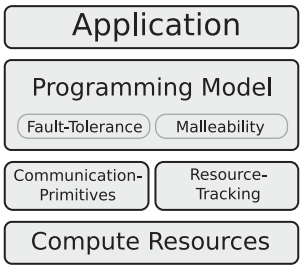

Figure 1. Abstract system hierarchy with resource tracking and communication primitives being the central low-level primitives for developing fault-tolerant and malleable programming models and applications.

available after the application has started. As a result, it is no longer possible to assume that resource allocation is static.

To run successfully in these increasingly dynamic environments, applications must be able to handle the inherent problems of these environments. Specifically, applications must incorporate both *malleability* [5], the capability to handle changes in the resources used during a computation, and *fault tolerance*, the capability to continue a computation despite failures. Without mechanisms for malleability and fault tolerance, the reliable execution of applications on dynamic systems is difficult, if not impossible.

A first step in creating a malleable and fault-tolerant system is to obtain an accurate and up-to-date view of the resources participating in a computation, and what roles they have. We therefore require some form of signaling whenever changes to the resource set occur. This information can then be used by the application itself, or by the runtime system (RTS) of the application's programming model, to react to these changes. In this article we refer to such functionality as *resource tracking*.

An important question is at what level in the software hierarchy should resource tracking be implemented. One option is to implement it in the application itself. However, this requires each application to implement resource tracking separately. Another option is to implement resource tracking in the RTS of the programming model of the application. Unfortunately, this still requires implementing resource tracking for each programming model separately. In addition, an implementation of resource tracking designed for use on a grid will be very different from one designed for a P2P environment. Therefore, the resource tracking functionality of each programming model will have to be implemented for each target environment as well. This situation is clearly not ideal.

Based on the observations above, we argue that resource tracking must be an integral part of a system designed for dynamic environments, *in addition* to the low-level communication primitives already present in such systems [6–8]. Figure 1 shows the position of resource tracking in a software hierarchy. There, a programming model's RTS uses low-level resource tracking functionality to implement the required higher level fault tolerance and malleability. In this way, resource tracking (indirectly) allows applications to run reliably and efficiently on dynamic systems, such as grids and clouds.

In this article we propose a general solution for resource tracking: the Join–Elect–Leave (JEL) model. JEL acts as an intermediate layer between the programming models and the environment they run on. As different environments have different characteristics, using a single implementation is impractical, if not impossible. Instead, several implementations of the JEL API are required, each optimized for a particular environment.

We have implemented JEL efficiently on clusters, grids, P2P systems, and clouds. These different JEL implementations can be used transparently by a range of programming models, in effect providing *unified* resource tracking for parallel and distributed applications across environments.

The contributions of this article are as follows:

- We show the need for unified resource tracking models in dynamic environments, such as grids, P2P systems, and clouds, and explore the requirements of these models.

DOI: 10.1002/cpe

- We define JEL: a unified model for tracking resources in dynamic environments. JEL is explicitly designed to be simple yet powerful, scalable, and flexible. The flexibility of JEL allows it to support parallel as well as distributed programming models.
- We show how JEL suits the resource tracking requirements of several programming models. We have implemented seven different programming models using JEL, ranging from traditional models, such as MPI-1 (in the form of MPJ [9]), to Satin [5], a high-level divide-and-conquer grid programming model that transparently supports malleability and fault tolerance.
- We show that JEL is able to function on a range of environments by discussing multiple implementations of JEL. These include a centralized solution for relatively stable environments, such as clusters and grids, and a fault-tolerant P2P implementation. In part, these implementations are based on the well-known techniques of information dissemination in distributed systems. Notably, JEL can be implemented efficiently in different environments, due to the presence of multiple consistency models.

Our research is performed in the context of the Ibis [7] Java-based grid computing project. In the previous work we presented the *Ibis Portability Layer* (*IPL*) [7], a communication library specifically targeted at dynamic systems such as grids. We augmented the IPL with our JEL resource tracking model, leading to a software system which can efficiently run applications on clusters, grids, P2P systems, and clouds. Using the software[§] developed in this project, including our implementations of JEL, we have been the first prize winner in a number of international competitions [10]. Notably, our winning submission in the Fault-Tolerant Category of the DACH 2008 Challenge[¶] Cluster/Grid 2008 in Tsukuba, Japan, made extensive use of the JEL model for detecting and reporting node failures.

This article is structured as follows. Section 2 discusses the requirements of a general resource tracking model. Section 3 shows one possible model fulfilling these requirements: our JEL model. Section 4 explains how JEL is used in several programming models. In Section 5 we discuss a (partially) centralized and a fully distributed implementation of JEL. Section 6 compares the performance of our implementations, and shows the applicability of JEL in real-world scenarios. As a worst case, we show that JEL is able to support even short-lived applications on large numbers of machines. Section 7 discusses the related work. Finally, Section 8 describes the future work and concludes.

2. REQUIREMENTS OF RESOURCE TRACKING MODELS

In this section we explore the requirements of resource tracking in a dynamic system. As mentioned above, resource tracking functionality can best be provided at a level between programming models and the computational environment (see Figure 1). A programming model's RTS uses this functionality to implement fault tolerance and malleability. This naturally leads to two sets of requirements for resource tracking: requirements imposed by the programming model above and requirements resulting from the environment below. We will discuss each in turn.

2.1. Programming model requirements

For any resource tracking model to be generally applicable, it needs to support multiple programming models, including both parallel and distributed models. Below is a list of the requirements covering the needs of most, if not all, parallel and distributed programming models.

List of participants: The most obvious requirement of a resource tracking model is the capability to build up a list of all the computational resources participating in a computation. When

[§] Implementations of programming models and other software referred to in this paper can be freely downloaded from `http://www.cs.vu.nl/ibis`.

[¶] `http://www.cluster2008.org/challenge/`.

DOI: 10.1002/cpe

communicating and cooperating with other participants of a computation, one must know who the other participants are.

Reporting of changes: Simply building a list of participants at startup is not sufficient. As resources may be added or removed during the runtime of a computation, a method for updating the current list of participants is also required. This can be done for instance by signaling the programming models' RTS whenever a change occurs.

Fault detection: Not all resources are removed gracefully. Machines may crash, and processes may be terminated unannounced by a scheduling system. For this reason, the resource tracking model also needs to include a failure detection and reporting mechanism.

Role selection: It is often necessary to select a leader from a set of resources for a specific task. For instance, a primary object may have to be selected in primary-copy replication, or a master may have to be selected in a master–worker application. Therefore, next to keeping track of which resources are present in a computation, a method for determining the roles of these resources is also required.

2.2. Environment requirements

Next to supporting multiple programming models, a generally applicable resource tracking model must also support multiple environments, including clusters, grids, clouds, and P2P systems. We now determine the requirements resulting from the environment in which a resource tracking model is used.

Small, simple interface: Different environments may have wildly different characteristics. On cluster systems, the set of resources is usually constant. On grids and clouds resource changes occur, albeit at a low rate. P2P systems, however, are known for their high rate of change. Therefore, different (implementations of) algorithms are needed for efficient resource tracking on different environments. To facilitate the efficient re-targeting of a resource tracking model, its interface must be as small and simple as possible.

Flexible quality of service: Even with a small and simple interface, it may not be possible to implement all the features of a resource tracking model efficiently on all environments with the same quality of service. For instance, *reliably* tracking *each and every* change to the set of resources in a small-scale cluster system is almost trivial, whereas in a large-scale P2P environment this is difficult to implement efficiently, if at all possible. However, not all programming models require the full functionality of a resource tracking model. Therefore, a resource tracking model should include quality of service features. If the resource tracking model allows for a programming model to specify the required features and their quality of service, a suitable implementation could be selected at runtime. This flexibility would greatly increase the applicability of a resource tracking model.

3. THE JOIN–ELECT–LEAVE MODEL

We will now describe our resource tracking model: JEL. JEL fulfills all the stated requirements of a resource tracking model. As shown in Figure 1, JEL is located at the same layer of the software hierarchy as low-level communication primitives. Applications use a programming model, ideally with support for fault tolerance and malleability. The programming model's RTS uses JEL for resource tracking, as well as a communication library. In this section we refer to programming models as *users* of JEL.

Figure 2 shows the JEL API. Next to an initialization function, the API consists of two parts: *Joins and Leaves*, and *Elections*. Together, these fulfill the requirements of parallel and distributed programming models as stated in the previous section.

In general, each machine used in a computation initializes JEL once, and is tracked as a single entity. However, modern machines usually contain multiple processors and/or multiple compute cores per processor. In some cases, it is therefore useful to start multiple processes per machine for a single computation, which then need to be individually tracked. In this paper, we therefore use

DOI: 10.1002/cpe

```
interface JEL {
    void init(Consistency electionConsistency,
              Consistency joinLeaveConsistency);
    void join(String poolName, Identifier identifier);
    void leave();
    void maybeDead(Identifier identifier);
    Identifier elect(String electionName);
    Identifier getElectionResult(String electionName);
}
//interface for notifications, called by JEL
interface JELNotifications {
    void joined(Identifier identifier);
    void left(Identifier identifier);
    void died(Identifier identifier);
}
```

Figure 2. JEL API (pseudocode, simplified).

the abstract term *node* to refer to a computational resource. Each node represents a single instance in a computation, be it an entire machine, or one processor of that machine.

JEL has been designed to work together with any communication library. The communication library is expected to create a unique identifier containing a contact address for each node in the system. JEL uses this address to identify nodes in the system, allowing a user to contact a node whenever JEL refers to it.

3.1. Joins and leaves

In JEL, the concept of a *pool* is used to denote the collection of resources used in a computation. To keep track of exactly which nodes are participating in a pool, JEL supports *join* notifications. Users are being notified whenever a new node joins a pool. When a node joins a pool, it also is notified of all nodes already present in the pool via the same notifications, given using the *JELNotifications* interface. This is typically done using callbacks, although a polling mechanism can be used instead, if callbacks are not supported by a programming language.

JEL also supports nodes leaving a computation, both gracefully and due to failures. If a node notifies JEL that it is leaving the computation, users of the remaining nodes in the pool receive a *leave* notification for this node. If a node does not leave gracefully, but crashes or is killed, the notification will consist of a *died* message instead. Implementations of JEL try to detect failing nodes, but the user can also report suspected failures to JEL using the *maybeDead* function.

3.2. Elections

It is often necessary to select a leader node from a set of resources for a specific task. To select a single resource from a pool, JEL supports *Elections*. Each election has a unique name. Nodes can nominate themselves by calling the *elect* function with the name of the election as a parameter. The identifier of the winner will be returned. Using the *getElectionResult* function, nodes can retrieve the result without being a candidate.

Elections are not democratic. It is up to the JEL implementation to select a winner from the candidates. For instance, an implementation may simply select the first candidate as the winner. At the user level, all that is known is that *some* candidate will be chosen. When a winner of an election leaves or dies, JEL will automatically select a new winner from the remaining living candidates. This ensures that the election mechanism will function correctly in a malleable pool.

3.3. Consistency models

Together, join/leaves and elections fulfill all resource tracking requirements of fault-tolerant and malleable programming models as stated in Section 2.1. However, we also require our model to be applicable to a wide range of environments, from clusters to P2P systems. To this end, JEL supports several consistency models for the join/leave notifications and the elections. These can be selected independently when JEL is initialized using the *init* function. Joins/leaves or elections

DOI: 10.1002/cpe

can also be turned off completely, if either part is not used. For examples of situations of when some parts of JEL remain unused, see Section 4.

Relaxing the consistency model allows JEL to be used on more dynamic systems such as P2P environments, where implementing strict consistency models cannot be done efficiently, if at all. For example, Section 5.2 describes a fully distributed implementation that is robust against failures, under a relaxed consistency model.

JEL offers two consistency models for joins and leaves. The *reliable* consistency model ensures that all notifications arrive in the same order on all nodes. Using reliable joins and leaves, a user can build up a list of all the nodes in the pool. As an alternative, JEL also supports *unreliable* joins and leaves, where notifications are delivered on a best effort basis, and may arrive out of order or not at all.

Similarly, JEL supports multiple consistency models for elections. If *uniform* elections are used, a *single* winner is guaranteed for each election, known at all nodes. Using the *non-uniform* model, an election is only guaranteed to converge to a single winner in unbounded time. The implementation of JEL will try to reach consensus on the winner of an election as soon as possible, but in a large system this may be time consuming. Before a consensus is reached, different nodes may perceive different winners for a single election. Intuitively, this non-uniform election has a very weak consistency. However, it is still useful in a number of situations (Section 4.2 shows an example).

4. APPLICABILITY OF JEL

JEL has been specifically designed to cover the required functionality of a range of programming models found in distributed systems. We have implemented JEL in the *IPL* [7], the communication library of the Ibis project. Figure 3 shows the position of JEL in the software stack of the Ibis project. All programming models implemented in the Ibis project use JEL to track resources, notably:

- Satin [5], a divide-and-conquer model
- Java RMI [11], an object-oriented RPC model
- GMI [12], a group method invocation model
- MPJ [9], a Java binding for MPI-1
- RepMI [12], a replicated object model
- Maestro [10], a fault-tolerant and self-optimizing dataflow model
- Jorus [10], a user-transparent parallel model for multimedia computing

As JEL is a generic model, it also supports other programming models. In addition to the models listed, we have implemented a number of prototype programming models, including data parallel, master–worker, and Bulk Synchronous Parallel (BSP) models. Although our current JEL implementations are implemented using Java, the JEL model itself is not limited to this language. The foremost problem when porting JEL to other programming languages is the possible absence of a callback mechanism. This problem can be solved by using downcalls instead. In addition,

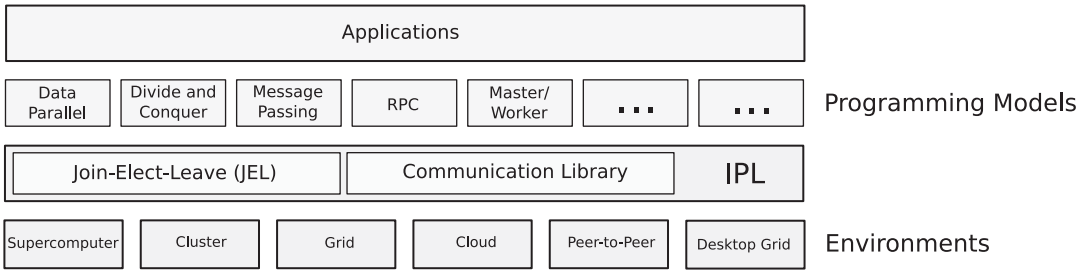

Figure 3. Position of JEL in the Ibis grid programming software stack.

DOI: 10.1002/cpe

Table I. Parts and consistency models of JEL used in the example programming models.

Model	Joins and leave	Elections
Master–Worker	—	Uniform
Divide-and-Conquer (elected master)	Unreliable	Uniform
Divide-and-Conquer (selected master)	Unreliable	Non-uniform
Message passing	Reliable	—

parts of current JEL implementations could be reused, for instance by combining the server of the centralized implementation with a client written in another language.

We will now illustrate the expressiveness of JEL by discussing several models in more detail. These programming models use different parts and consistency models of JEL, see Table I for an overview.

4.1. Master–worker

The first programming model we discuss is the *master–worker* [13] model, which requires a single node to be assigned as the *master*. As the master controls the application, its identity must be made available to all other (*worker*) nodes. Depending on the application, the number of suitable candidates for the role of master may range from a single node to all participating nodes. For this selection, the master–worker model uses uniform elections.

As the workers do not intercommunicate, the only information that a worker needs in a master–worker model is the identity of the master node. Thus, in this model, joins and leaves are not needed and can simply be switched off.

4.2. Divide-and-conquer

The second programming model that we discuss is divide-and-conquer. We use Satin [5] as an example of such a system. Satin is malleable, can handle failures, and hides many intricacies of the grid from the application programmer. It also completely hides which resources are used. Distribution and load balancing are performed automatically by using random work stealing between nodes. Satin is *cluster-aware*: it exploits the hierarchical nature of grids to optimize load balancing and data transfer. For instance, nodes prefer to steal work from nodes inside their local cluster, as opposed to from remote sites. The Satin programming model requires support from the resource tracking model for adding new nodes, as well as removing running nodes (either gracefully or due to a crash). Satin applies this information to re-execute subtasks if a processor crashes. In addition, it dynamically schedules subtasks on new machines that become available during the computation, and it migrates subtasks if machines leave the computation.

Although Satin requires notifications whenever nodes join or leave the computation, these notifications do not need to be completely reliable, nor do they need to be ordered in any way. Satin uses the joins and leaves to build up a list of nodes in the pool. This list is then used to randomly select nodes to steal work from. As long as each node has a reasonably up-to-date view of who is participating in the application, Satin will continue to work. When the information is out-of-date or incomplete, the random sampling will be skewed slightly, but in practice the negative impact on the performance is small (see Section 6.4). Satin therefore uses the *unreliable* consistency of the join and leave notifications.

In Satin, an election is used to select a special coordinator per cluster. These coordinators are used to optimize the distribution of fault tolerance-related data in wide-area systems. When multiple coordinators are present, more data will be transferred, which may lead to a lower performance. Satin will still function correctly, however. Therefore, the election mechanism used to select the cluster coordinators does not necessarily have to return a unique result, meaning that the *non-uniform* elections of JEL can be used.

When an application is starting, Satin needs to select a master node that starts the main function of the application. This node can be explicitly specified by the user or application, or it can be

DOI: 10.1002/cpe

automatically selected by Satin. The latter requires the *uniform* election mechanism of JEL. If the master node is specified in advance by the user, no election is needed for this functionality.

From the discussion above, we can conclude that the requirements of Satin differ depending on the circumstances. If the user has specified a master node, Satin requires *unreliable* join and leave notifications for the list of nodes, as well as *non-uniform* elections for electing cluster coordinators. If, on the other hand, a master node must be selected by Satin itself, *uniform* elections are an additional requirement.

4.3. Message passing (MPI-1)

The last programming model we discuss is the Message Passing model, in this case represented by the commonly used MPI [6] system. MPI is widely used on clusters and even for multi-site runs on grid systems. We implemented a Java version of MPI-1, MPJ [9]. The MPI model assigns ranks to all nodes. Ranks are integers uniquely identifying a node, assigned from 0 up to the number of nodes in the pool. In addition, users can retrieve the total number of nodes in the system.

Joins and leaves with reliable consistency are guaranteed to arrive in the same order on all nodes. This allows MPI to build up a totally ordered list of nodes, by assigning rank 0 to the first node that joins the pool, rank 1 to the second, etc. As in the master–worker model, MPI does not require all functionalities of JEL, as elections are not used.

MPI-1 has very limited support for changes of resources and failures. Applications using this model cannot handle changes to the resources, such as nodes leaving or crashing. Using an MPI implemented on top of JEL will not fix this problem. However, some extensions to MPI are possible. For instance, MPI-2 supports new nodes joining the computation, Phoenix [14] adds supports for nodes leaving gracefully, and FT-MPI [15] allows the user to handle faults, by specifying the action to be taken when a node dies. All these extensions to MPI can be implemented using JEL for the required resource tracking capabilities.

5. JEL IMPLEMENTATIONS

It is impractical, if not impossible, to use the same implementation of JEL on clusters, grids, clouds, as well as P2P systems. As these different environments have different characteristics, there are different trade-offs in implementation design. We have explored several alternative designs, and discuss these in this section.

On cluster systems, resources used in a computation are mostly fixed, and do not change much over time. Therefore, our JEL implementation targeted at single cluster environments uses a relatively simple algorithm for tracking resources, based on a central coordinator. This ensures high performance and scalability, and the simple design leads to a more robust, less error-prone implementation. This *central* implementation provides reliable joins and leaves, and uniform elections. As this implementation uses a central coordinator for tracking resources, these stronger consistency models can be implemented without much effort.

On more dynamic systems, such as grids, clouds, and desktop grids, the simple implementation design used on clusters is not sufficient. As the number of machines in the system increases, so does the number of failures. Moreover, any change to the set of resources needs to be disseminated to a larger set of machines, possibly with high network latencies. Thus, these environments require a more *scalable* implementation of JEL. We used a number of techniques to decrease the effort required and the amount of data transferred by the central coordinator, at the cost of an increased complexity of the implementation. As the resource tracking still uses a central coordinator, the stronger consistency models for joins, leaves, and elections of JEL are still available.

Lastly, we implemented JEL on P2P environments. By definition, it is not possible to use centralized components in P2P systems. Therefore, our P2P implementation of JEL is fully distributed. Using Lamport clocks [16] and a distributed election algorithm [17] it is possible to implement strong consistency models in a fully distributed manner. However, these algorithms are prohibitively

DOI: 10.1002/cpe

difficult to implement. Therefore, our P2P implementation only provides unreliable joins and leaves, and non-uniform elections, making it extremely simple, robust, and scalable. We leave implementing a P2P version of JEL with strong consistency models for future work.

As mentioned above, we have augmented our IPL [7] with JEL. The IPL is a low-level message-based communication library implemented in Java, with support for streaming and efficient serialization of objects. All functionalities of JEL are exported in the IPL's *Registry*. JEL is implemented in the IPL as a separate thread of the Java process. Notifications are passed to the programming models' RTS or to application using a callback mechanism.

5.1. Centralized JEL implementation

Our centralized JEL implementation uses a single server to keep track of the state of the pool. Using a centralized server makes it possible to implement stronger consistency models. However, it also introduces a single point of failure and a potential performance bottleneck.

The server has three functions. First, it handles requests of nodes participating in the computation. For example, a node may signal that it has joined the computation, is leaving, or is running for an election. By design, these requests require very little communication or computation.

Second, the server tracks the current resources in the pool. It keeps a list of all nodes and elections, and detects failed nodes. Our current implementation is based on a *leasing* mechanism, where nodes are required to periodically contact the server. If a node has had no contact with the server for a certain number of seconds, it sends a so-called *heartbeat* to the server. If it fails to do so, the server will try to connect to the node, to see if the node is still functional. If the server cannot reach the node, this node is declared *dead*, and removed from the pool.

Third, the server disseminates all changes of the state of the pool to the nodes. The nodes use these updates to generate join, leave, died, and election notifications for the application. If there are many nodes, the dissemination may require a significant amount of communication and lead to performance problems. To alleviate these problems we use a simple yet effective technique. Any changes to the state of the pool are mapped to *events*. These events have a *unique sequence number*, and are *totally ordered*. An event represents a node joining, a node leaving, a node dying, or an election result.

A series of state changes to a sequence of events can now be perceived as a *stream* of events. Dissemination of this stream can be optimized using well-known techniques, such as broadcast trees or gossiping. Figure 4 shows an example of a stream of events. In this case, two nodes join, one leaves, one is elected master, and then dies. This stream of events thus results in an empty pool.

We have experimented with four different methods of disseminating the event stream: a simple serial send, serial send with peer bootstrap, a broadcast tree, and gossiping. The different mechanisms and their implementations are described below.

5.1.1. Serial send. In our first dissemination technique, the central server forwards all events occurring in the pool to each node individually. Such a serial send approach is straightforward to implement, and is very robust. It may lead to performance problems though, as a large amount of data may have to be sent by the server. To optimize the network usage, the server sends to multiple nodes concurrently.

In this implementation, a large part of the communication performed by the server consists of sending a list of all the nodes to a new, joining node (the so-called *bootstrap* data). If many nodes join a computation at the same time, this may cause the server to become overloaded.

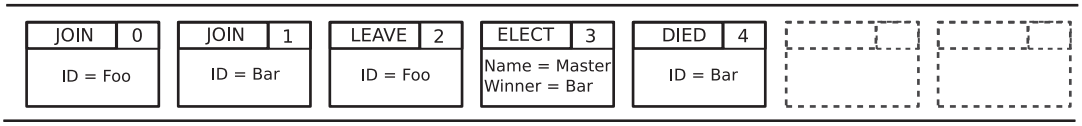

Figure 4. Example of an event stream.

DOI: 10.1002/cpe

5.1.2. Peer bootstrap. As an optimization of the serial send technique, we implemented *peer bootstrapping*, where joining nodes use other nodes (their *peers*) to obtain the necessary bootstrap data. When a node joins, the server sends it a small list of randomly chosen nodes in the pool. The joining node then tries to obtain the bootstrap data from the nodes in this list. If, for some reason, none of the nodes in the list can be reached, the joining node uses the server as a backup source of bootstrap data. This approach guarantees that the bootstrap process will succeed eventually.

5.1.3. Broadcast tree. A more efficient way of disseminating the stream of events from the server to all nodes is a broadcast tree. Broadcast trees limit the load on the server by using the nodes themselves to forward data. Broadcast trees also have disadvantages, as the tree itself is a distributed data structure that needs to be managed. This requires significant effort, and makes broadcast trees less robust than serial send.

Our broadcast implementation uses a binomial tree structure with the server as the root of the tree, which is also commonly used in MPI implementations [18]. To minimize the overhead of managing the tree, we use the data stream being broadcast to manage the tree. As this stream includes totally ordered notifications of all joining and leaving nodes, we can use it to construct the broadcast tree at each node.

To increase the robustness of our broadcast implementation, we implemented *fallback* information dissemination. Periodically, the server directly connects to each node in the pool, and sends it any events it has not received yet. This fallback mechanism guarantees the functioning of the system, regardless of the number, and type, of failures occurring. In addition, it causes very little overhead if there are no failures.

5.1.4. Gossiping. A fourth alternative for disseminating the events of a pool to all its nodes is the use of *gossiping* techniques. Gossiping works on the basis of periodic information exchanges (gossips) between peers (nodes). Gossiping is robust, easy to implement, and has low resource requirements.

In the gossiping dissemination, all nodes record the event stream. Periodically, a node contacts one of its peers. The event stream of those two nodes are then merged by sending any missing events from one peer to the other. To reduce memory usage old events are eventually purged from the system.

Although the nodes exchange events among themselves, the pool is still managed by the central server. The server still acts as a contact point for nodes that want to join, leave, or run for an election. In addition the server creates all events, determines the ordering of events, detects failing nodes, etc.

To seed the pool of nodes with data, the server periodically contacts a random node, and sends it any new events. The nodes will then distribute these new events among themselves using gossiping. When the nodes gossip at a fixed interval, the events travel through the system at an exponential rate. The dissemination process thus requires a time that is logarithmically proportional to the pool size.

To speed up the dissemination of the events to all nodes, we implemented an *adaptive* gossiping interval at the server. Instead of waiting a fixed time between sending events to nodes, we calculate the interval based on the size of the pool by dividing the standard interval by the base 2 logarithm of the pool size. Thus, events are seeded at a speed proportional to the pool size. The dissemination speed of events becomes approximately constant, at the expense of an increase in communication load on the server.

As gossip targets are selected randomly, there is no guarantee that all nodes will receive all events. To ensure reliability, we use the same fallback dissemination technique that we used in the broadcast tree implementation. Periodically, the server contacts all nodes and sends them any events they do not have.

5.2. Distributed JEL implementation

Although the performance problems of the centralized implementation are largely solved by using broadcast trees and gossiping techniques, the server component is still a central point of failure,

DOI: 10.1002/cpe

and not suitable for usage in P2P systems. As an alternative, we created a fully distributed implementation of JEL using P2P techniques. It has no central components, hence failures of individual nodes do not lead to a failure of the entire system.

Our implementation is based on our ARRG [19] gossiping algorithm. ARRG is resilient against failures, and can handle network connectivity problems, such as firewalls and Network Address Translations (NATs). Each node in the system has a unique identifier in the form of a UUID [20], which is generated locally at startup. ARRG needs the address of an existing node at startup to bootstrap, hence this must be provided. This address is used as an initial contact point in the pool. ARRG provides a so-called *peer sampling service* [21], guaranteeing a random sampling of the entire pool even if failures and network problems occur.

In addition to ARRG, we use another gossiping algorithm to exchange data on nodes and elections. Periodically, a node connects to a random node (provided by ARRG) and exchanges information about other nodes and elections. It sends a random subset of the nodes and elections it knows and includes information about itself. It then receives a number of members and elections from the peer node, and merges these with its own state. Over time, the nodes build up a list of the nodes and elections in the pool.

If a node wants to leave the computation, it sends out this information to a number of nodes in the system. Eventually, this information will reach all nodes. As a crashed node cannot send a notification to the other nodes indicating that it has died, a distributed failure detection mechanism is needed.

The failure detection mechanism uses a *witness* system. A timeout is kept in every entry on a node, indicating the last time that this node has successfully been contacted. Whenever the timeout expires, a node is suspected of having died. Nodes with expired entries in their node list try to contact these suspects. If this fails, they add themselves as a witness to this node's demise. The witness list is part of the gossiped information. If a sufficient number of nodes declare that a node has died, it is pronounced dead.

Besides joins and leaves, the distributed implementation also supports elections. Because of the difficulties of implementing distributed election algorithms [17], and the lack of guarantees even when using the more advanced algorithms, we only support the non-uniform election consistency model. In this model, an election converges to a single winner. Before that time, nodes may not agree on the winner of that election.

Election results are gossiped. When a node needs the result of an unknown election, it simply declares itself as the winner. If a conflict arises when merging two different election results, one of the two winners is selected deterministically (the node with the numerically lowest UUID wins). Over time, only a single winner remains in the system.

As a consequence of the aforementioned design, the distributed implementation of JEL is fault tolerant in many aspects. First, the extensive use of gossiping techniques inherently leads to fault tolerance. The ARRG protocol adds further tolerance against failures, for example by using a fallback cache containing previously successful contacts [19]. Most importantly, the distributed implementation lacks *any* centralized components, providing fully distributed implementations of all the required functionality instead.

6. EVALUATION

To evaluate the performance and scalability of our JEL implementations, we performed several experiments. These include low-level and application-level tests on multiple environments. In particular, we want to assess how much the performance is sacrificed to gain the robustness of a fully distributed implementation, as we expect this implementation to have the lowest performance. Exact quantification of performance differences between implementations, however, is difficult—if not impossible. As shown below, the performance results are highly dependent on the characteristics of the underlying hardware. Furthermore, the impact on application performance, in turn, is dependent on the programming model used. For example, MPI cannot proceed until all nodes have joined,

DOI: 10.1002/cpe

whereas Satin starts as soon as a resource is available. All the experiments were performed multiple times. The numbers shown are taken from a single representative experiment.

6.1. Low-level benchmark: join test

The first experiment is a low-level stress test using a large number of nodes. We ran the experiment on two different clusters. The purpose of the experiment is to determine the performance of our JEL implementations under different network conditions. In the experiment, all the nodes join a single pool and, after a predetermined time, leave again. As a performance metric, we use the *average perceived pool size*. To determine this metric, we keep track of the pool size at all nodes. Ideally, this number is equal to the actual pool size. However, if a node has not received all notifications, the perceived pool size will be smaller. We then calculate the average perceived pool size over all nodes in the system. The average is expected to increase over time, eventually becoming equal to the actual pool size. This indicates that all the nodes have received all the notifications. The shorter the stabilization time, the better.

This experiment was done on our DAS-2 and DAS-3 clusters. The DAS-2 cluster consists of 72 dual processor Pentium III machines, with 2Gb Myrinet interconnect. The DAS-3 cluster consists of 85 dual-CPU dual-core Opteron machines, with 10Gb Myrinet. See http://www.cs.vu.nl/das2 and http://www.cs.vu.nl/das3 for more information.

As neither the DAS-2 nor the DAS-3 have a sufficiently large number of machines to stress test our implementation, we started multiple nodes per machine. As neither our JEL implementations or the benchmark are CPU bound, the sharing of CPU resources does not influence our measurements. The nodes do share the network bandwidth though. However, all implementations of JEL are affected equally, hence the relative results of all tested implementations remain valid. The server of the centralized implementation of JEL is started on the front-end machine of the cluster.

6.1.1. DAS-2. Figure 5 shows the performance of JEL on the DAS-2 system. We started 10 nodes per processor core on 50 dual processor machines, for a total of 1000 nodes. Owing to the sharing of network resources, all nodes, as well as the frontend running the server, have an effective bandwidth of about 100 Mbit/s.

For convenience, we only show the first 100 s of the experiment, when all the nodes are joining. The graph shows that the serial send dissemination suffers from a lack of network bandwidth, and is the lowest performing implementation.

The peer bootstrap and broadcast tree techniques perform equally well on this system. This is not surprising, as the broadcast tree and peer bootstrap techniques utilize all nodes to increase throughput. As the graph shows, adaptive gossip dissemination is faster than the normal central gossip version, as it adapts its speed to the pool size.

While not shown in the graph, the fully distributed implementation is also converging to the size of the pool, albeit slower than most versions of the centralized implementation. The slow speed is caused by an overload of the bootstrap service, which receives 1000 gossip requests within a few milliseconds when all the nodes start. This is an artifact of this artificial test that causes

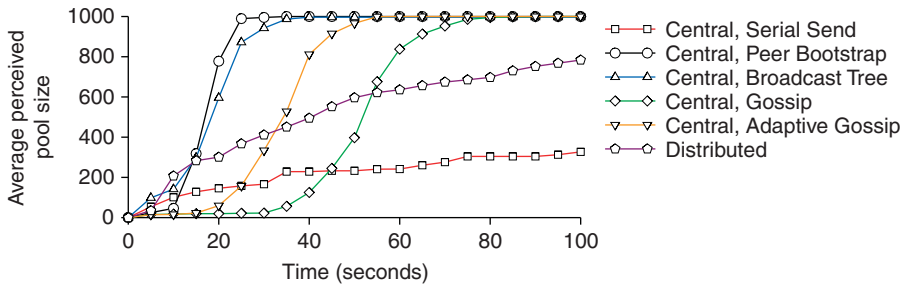

Figure 5. 1000 nodes Join test (DAS-2).

DOI: 10.1002/cpe

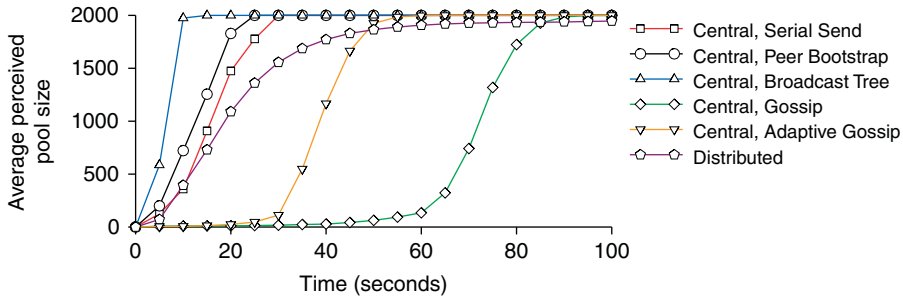

Figure 6. 2000 nodes Join test (DAS-3).

Table II. Total data transferred in Join test with 2000 nodes on the DAS-3.

Implementation	Dissemination	Server (MB)	Node average (MB)
Central	Serial send	1521.47	0.76
	Peer bootstrap	677.23	0.45
	Broadcast tree	5.57	1.32
	Gossip	9.83	0.49
	Adaptive gossip	40.36	0.57
Distributed	Gossip	n.a.	25.37

all the nodes to start simultaneously. In a P2P environment this is unlikely to occur. Multiple instances of the bootstrap service would solve this problem. Yet, the performance of the distributed implementation is acceptable, especially considering the high robustness of this implementation.

6.1.2. DAS-3. Next, we examine the performance of the same benchmark on the newer DAS-3 system (see Figure 6). As a faster network is available on this machine, congestion of the network is less likely. As the DAS-3 cluster has more processor cores, we increased the number of nodes to 2000, resulting in 250 Mbit/s of bandwidth per node. The frontend of our DAS-3 cluster has 10 Gbit/s of bandwidth. Performance on the DAS-3 increases significantly compared to the DAS-2, mostly because of the faster network. The serial send and gossip techniques no longer suffer from network congestion at the server or bootstrap service. As a result, the performance increases dramatically for both. In addition, the graph shows that the performance of the broadcast tree is now significantly better than any other dissemination technique.

The performance of the central implementation with gossiping is influenced by the larger size of the pool. It takes considerably longer to disseminate the information to all nodes. As before, the adaptive gossiping manages to adapt, and reaches the total pool size significantly faster.

From our low-level benchmark on both the DAS-2 and DAS-3 we conclude that it is possible to implement JEL such that it is able to scale to a large number of nodes. In addition, a number of different implementation designs are possible for JEL, all leading to reasonable performance.

6.2. Network bandwidth usage

To investigate the cost of using JEL, we recorded the total data transferred by both the server and the clients in the previous experiment. Table II shows the total traffic generated by the experiment on DAS-3, after all the nodes have joined and left the pool.

Using the serial send version, the server transferred over 1500 MB in the 10 min experiment. Using peer bootstrap already halves the traffic needed at the server. However, the broadcast tree dissemination uses less than 5 MB of server traffic to accomplish the same result. It does this by using the nodes of the system, leading to a slightly higher traffic at the nodes (1.32 MB instead of 0.76 MB).

DOI: 10.1002/cpe

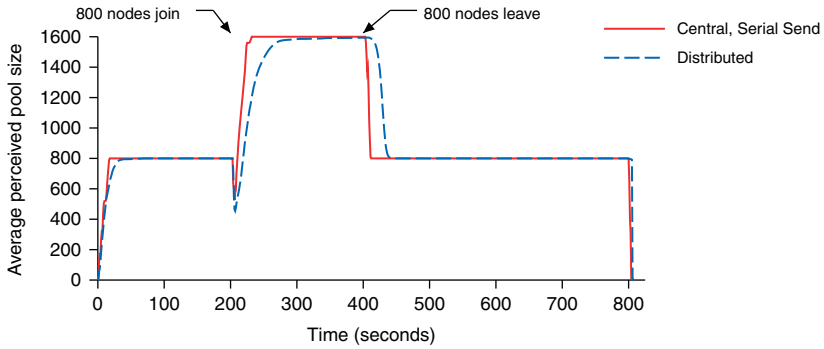

Figure 7. Join/leave test run on four clusters across the DAS-3 grid. Half of the nodes only start after 200 s, and leave after 400 s.

From this experiment we conclude that the dissemination techniques significantly increase the scalability of our implementation. Also, the broadcast tree implementation is very suited for low bandwidth environments. For the distributed implementation, the average traffic per node is 25 MB, an acceptable cost for having a fully distributed implementation.

6.3. Low-level benchmark in a dynamic environment

We now test the performance of JEL in a dynamic environment, namely the DAS-3 grid. Besides the cluster at the VU used in the previous tests, the DAS-3 system consists of four more clusters across the Netherlands. For this test we started our Join benchmark on two clusters (800 nodes), and add two clusters later, for a total of 1600 nodes. Finally, two clusters also leave, either gracefully, or by *crashing*.

Results of the test when the nodes leave gracefully are shown in Figure 7. We tested both the central implementation of JEL and the distributed implementation. For the central implementation we have selected the serial send dissemination technique, which performs average on DAS-3 (see Figure 6). On the scale of the graph of Figure 7 results obtained for the other techniques are indistinguishable.

Figure 7 shows that both implementations are able to track the entire pool. As said, the pool size starts at 800 nodes, and increases to 1600 nodes 200 s into the experiment. The *dip* in the graph at 200 s is an artifact of the metric used: At the moment 800 extra nodes are started, these nodes have a perceived pool size of 0. Thus, the average over all nodes in the pool halves. As in the previous test, the central implementation is faster than the distributed implementation. After 400 s, two of the four clusters (800 of the 1600 nodes) leave the pool. The graph shows that JEL correctly handles nodes leaving, with both implementations processing the leaves shortly.

As said, we also tested with the nodes crashing by forcibly terminating the node's process. The results can be seen in Figure 8. When nodes crash instead of leaving, it takes longer for JEL to detect these nodes have died. This delay is due to the timeout mechanism in both implementations. A node is only declared dead if it cannot be reached for a certain time (a configuration property of the implementations, in this instance set to 120 s). Thus, nodes are declared dead with a delay after crashing. The central implementation of JEL has a slightly longer delay, as it tries to contact the faulty nodes one more time after the timeout expires. From this benchmark we conclude that JEL is able to function well in dynamic systems, with both leaving and failing nodes.

6.4. Satin gene sequencing application

To test the performance of our JEL implementations in a real world setting, we used 256 cores of our DAS-3 cluster to run a gene sequencing application implemented in Satin [5]. Pairwise sequence alignment is a bioinformatics application where DNA sequences are compared with each other to identify similarities and differences. We run a large number of instances of the well-known

DOI: 10.1002/cpe

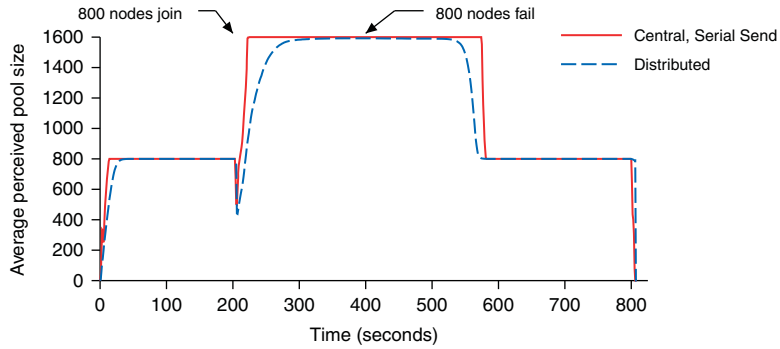

Figure 8. Join/fail test run on four clusters across the DAS-3 grid. Half of the nodes only start after 200 s, and crash after 400 s.

Table III. Gene sequencing application on 256 cores of the DAS-3.

		Run time		
Implementation	Dissemination	Small	Large	Join time
Central	Serial send	71.7	408.0	18.2
	Peer bootstrap	70.5	406.1	17.2
	Broadcast tree	66.4	402.9	10.6
	Gossip	67.7	426.6	14.6
	Adaptive gossip	67.5	426.4	11.1
Distributed	Gossip	82.3	462.4	14.1

Listed are total runtime (in seconds) of the application for two problem sizes and time (in seconds) until all nodes have joined fully (average perceived pool size is equal to the actual pool size). Runtime includes the join time.

Smith-Waterman [22] algorithm in parallel using Satin's divide-and-conquer programming style. The resulting application achieves excellent performance (93%efficiency on 256 processors).

Table III lists the performance of the application for various JEL implementations, and two different problem sizes. We specifically chose to include a small problem on a large number of cores to show that our JEL implementations are also suitable for short-running applications where the overhead of resource tracking is relatively large. In this very small problem, the application only ran for little over a minute. The table shows similar performance for all versions of JEL. Moreover, the relative difference is even smaller in the large problem size. An exception are the implementations based on gossiping techniques. The periodic gossiping causes a small but constant amount of network traffic. Unfortunately, the load balancing mechanism of Satin is very sensitive to this increase in network load. Though the distributed implementation lacks the guaranteed delivery of notifications present in the central implementation, Satin is able to perform the gene sequencing calculations with only minor delay. This is an important result, given Satin's transparent support for malleability and fault tolerance, as explained in Section 4.2.

To give an impression of the overhead caused by JEL, we also list the *join time*, the amount of time from the start of the application it takes for the average perceived pool size to reach the actual pool size, i.e. the time JEL needs to notify all nodes of all joins. The join time of an application is independent of the runtime of the application, and mainly influenced by number of nodes, JEL implementation, and resources used. Therefor, we only list the join time once, for both problem sizes. The performance of the various JEL implementations is in line with the low-level benchmark results, with the broadcast tree implementation being the fastest. Our gene sequencing experiment shows that our model and implementations are able to handle even these short running applications.

DOI: 10.1002/cpe

Table IV. Sites used in the worldwide divide-and-conquer experiment.

Location	Country	Type	Nodes	Cores	Efficiency (%)
VU University, Amsterdam	The Netherlands		32	128	97.3
University of Amsterdam		Grid	16	64	96.5
Delft University		(DAS-3)	32	64	94.0
Leiden University			16	32	96.7
National Institute of Informatics, Chiba	Japan	Grid	8	16	84.0
University of Tsukuba		(InTrigger)	8	64	81.1
VU University, Amsterdam	The Netherlands	Desktop Grid	16	17	98.0
Amazon EC2	USA	Cloud	16	16	93.2
		Total	176	401	94.4

Efficiency is calculated as the difference between total runtime of the application process and time spent computing. Overhead includes joining and leaving, as well as application communication for load balancing, returning results, etc.

6.5. Worldwide experiment

To show that JEL is suitable for a large number of different environments, we performed a worldwide experiment using the central implementation of JEL with serial send dissemination. We used a prototype of the pending re-implementation of Satin, especially designed for limited connectivity environments. In our worldwide experiment, connectivity between sites is often limited because of firewalls, and the network includes a number of low bandwidth and high latency links.

As an application we used an implementation of *First Capture Go*, a variant of the *Go* board game where a win is completed by capturing a single stone. Our application determines the optimal move for a given player, given any board. It uses a simple brute-force algorithm for determining the solution, trying all possible moves recursively using a divide-and-conquer algorithm. Since the entire space needs to be searched to calculate the optimal answer, our application does not suffer from search overhead.

Table IV shows an overview of the sites used. These consist of two grids (the DAS-3 in the Netherlands and the InTrigger [23] system in Japan), a desktop grid consisting of student PCs at the VU University Amsterdam, and a number of machines in the Amazon EC2 [4] compute cloud in the U.S.A. We used a total of 176 machines, with a total of 401 cores. As we started a single process per machine, and used threads to distribute work among cores, this amounts to 176 JEL nodes.

Figure 9 shows the communication structure of the experiment. The graph shown is produced by the visualization of the SmartSockets [24] library, which is used to connect all the nodes despite of the firewalls present. In the graph, each site is represented by a different color. Next to the compute nodes themselves (called *Instances* in the graph), and the central server, a number of support processes is used. All part of the SmartSockets [24] library, these support processes allow communication to pass through firewalls, monitor the communication, and produce the visualization shown. The support processes run on the frontend machines of the sites used.

Our worldwide system finishes the capture Go application in 35 min. We measured the efficiency of the machines, comparing the total time spent computing to the total runtime of the processes. Overhead includes joining and leaving, as well as time spent communicating with other nodes to load balance the application, return results, etc. Efficiency of the nodes ranges from 79.8 to 99.1%. The low efficiency on some nodes is due to the severely limited connectivity of these nodes: the nodes of the InTrigger grid in Japan can only communicate with the outside world through an ssh tunnel, with a bandwidth of only 1Mbit/s and a latency of over 250 ms to the DAS-3. Even with some nodes having a somewhat diminished efficiency, the average efficiency over all the nodes in the worldwide experiment is excellent, at 94.4%.

Although JEL adds to the overhead of the application, running the experiment without JEL would be difficult, if not impossible. Without JEL, all nodes would have to be known before starting the application, and this list would have to be spread manually to all nodes. In addition, the connectivity problems of the InTrigger grid in Japan lead to these nodes starting the computation

DOI: 10.1002/cpe

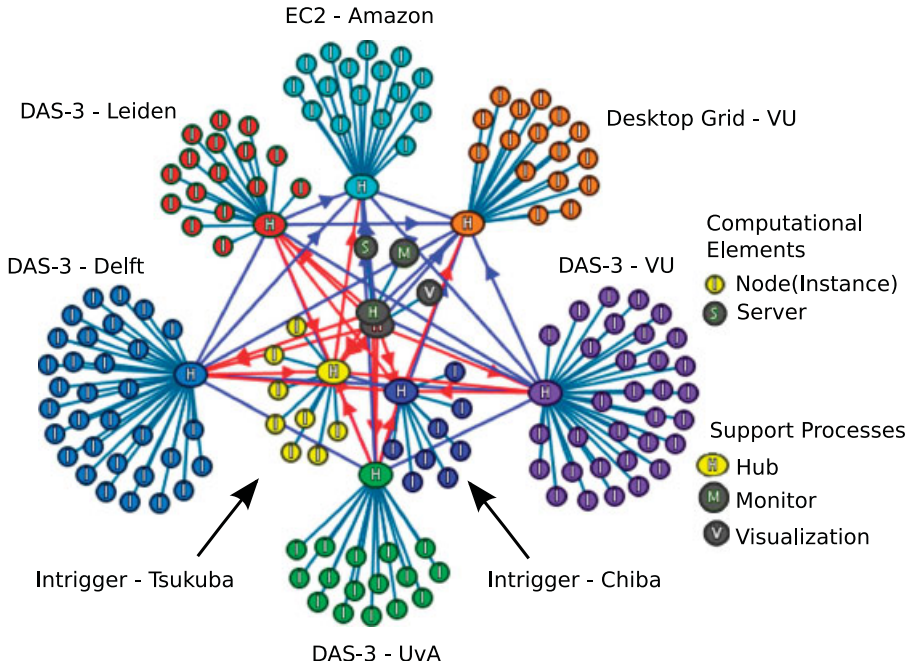

Figure 9. Communication structure of the worldwide divide-and-conquer experiment. Nodes in this graph represent processes, edges represent connections. The experiment contains both nodes performing the computation, as well as a number of support processes which allow communication to pass through firewalls, monitor the communication, and produce this image. Each color represents a different location.

with a significant delay. With JEL, these nodes simply join the running computation later, when the rest of the nodes have already done a significant amount of work. Our experiment shows that JEL is suitable for running applications on a large scale and a wide range of systems, including desktop grids and clouds.

6.6. Competitions

Recently, the software produced by the Ibis project (which includes JEL as one of its core components) has been put to the test in two international competitions [10] organized by the IEEE Technical Committee on Scalable Computing, as part of the CCGrid 2008 (Lyon, France) and Cluster/Grid 2008 (Tsukuba, Japan) international conferences.

The first competition we participated in was SCALE 2008, or the First IEEE International Scalable Computing Challenge. Our submission consisted of a multimedia application, which is able to recognize objects from webcam images. These images are sent to a grid for processing, and the resulting image descriptions are used to search for objects in a database. In our application, JEL is used to keep track of precisely which grid resources are available for processing images.

The second competition was DACH 2008, or the First International Data Analysis Challenge for Finding Supernovae. Here, the goal was to find 'supernova candidates' in a large distributed database of telescope images. Again, we used JEL in our submission to keep track of all the available resources.

The DACH challenge consisted of two categories: a *Basic Category* where the objective was to search the entire database as fast as possible and a *Fault-Tolerant* category, where next to speed, fault tolerance was also measured by purposely killing over 30% of the nodes in the computation. Especially in the Fault-Tolerant category, JEL was vital for the successful completion of the application.

Using our software (including JEL), we have won first prize in both SCALE 2008 and DACH 2008. Moreover, we won both the Basic and the Fault-Tolerant categories at DACH 2008. These

DOI: 10.1002/cpe

prizes show that JEL is very effective in many real-world scenarios, including dynamic systems with failing nodes.

7. RELATED WORK

Other projects have investigated supporting malleability and fault tolerance in various environments, and resource tracking in these systems. However, most of these projects focus on a single programming model and a single target environment.

One area of active research for supporting applications on more dynamic environments is the MPI standard. As said, the MPI-1 standard does not have support for nodes joining or leaving the computations. To alleviate this problem the follow-up MPI-2 [6] standard also supports changes to the nodes in a system. A process may *spawn* new instances of itself, or connect to a different running set of MPI-2 processes. A very basic naming service is also available.

Although it is possible to add new processes to an MPI application, the resource tracking capabilities of MPI-2 are very limited by design and a MPI implementation is not required to handle node failures. In addition, notifications of changes such as machines joining, leaving, or crashing are not available. Thus, resource tracking of MPI-2 is very limited, unlike our generic JEL model.

One MPI derivative that does offer explicit support for fault tolerance is FT-MPI [15]. FT-MPI extends the MPI standard with functionality to *recover* the MPI library and runtime environment after a node fails. In FT-MPI, an application can specify if failed nodes must be simply removed (leaving gaps in the ranks used), replaced with new nodes, or if the groups and communicators of MPI must be *shrunk* so that no gap remains. Recovering the application must still be done by the application itself.

FT-MPI relies on the underlying system to detect failures and notify it of these failures. The reference implementation of FT-MPI uses HARNESS [25], a distributed virtual machine with explicit support for adding and removing hosts from the virtual machine, as well as failure detection. HARNESS shares much of the same goals as JEL, and is able to overcome many of the same problems JEL tries to solve. However, HARNESS focuses on a smaller set of applications and environments than JEL. HARNESS does not explicitly support distributed applications, as JEL does. Also, HARNESS does not offer the flexibility to select the concurrency model required by the application, hindering the possibility for more loosely coupled implementations of the model, such as the P2P implementation of JEL.

Other projects have investigated supporting dynamic systems. One example is Phoenix [14], where an MPI-like message passing model is used. This model is extended with support for *virtual nodes*, which are dynamically mapped to *physical nodes*, the actual machines in the system. GridSolve [26] is a system for using resources in a grid based on a *client–agent–server* architecture. The 'View Synchrony' [27] shared data model also supports nodes joining, leaving and failing. Again, all these programming models focus on resource tracking for a single model, not the generic resource tracking functionality offered by JEL. All models mentioned can be implemented using the functionality of JEL.

Although all our current JEL implementations use gossiping and broadcast trees as a means for information dissemination, other techniques exist. One example is the publish-subscribe model [28]. Despite the fact that information dissemination is an important part of JEL, our model offers much more functionality to provide a full solution for the resource tracking problem. Most importantly, further functionality includes the *active* creation and gathering of information regarding (local) changes in the resource set.

All current implementations of JEL are build from the ground up, with little external dependencies. However, JEL implementations could in principal interface with external systems, for instance Grid Information Services (GIS [29]). These systems can be used both for acquiring (monitoring) data, as well as disseminating the resulting information. One key difference between JEL and current monitoring systems is the fact that JEL tracks resources of *applications*, not *systems*. An application crashing usually does not cause the entire system to cease

DOI: 10.1002/cpe

functioning. Sole reliance of system monitoring data will therefore not detect application-level errors.

8. CONCLUSIONS AND FUTURE WORK

With the transition from static cluster systems to dynamic environments, such as grids, clusters, clouds, and P2P systems, fault tolerance and malleability are now essential features for applications running in these environments. A first step in creating a fault-tolerant and malleable system is *resource tracking*: the capability to track exactly which resources are part of a computation and what roles they have. Resource tracking is an essential feature in any dynamic environment, and should be implemented on the same level of the software hierarchy as communication primitives.

In this paper we presented JEL: a unified model for tracking resources. JEL is explicitly designed to be scalable and flexible. Although the JEL model is simple, it supports both traditional programming models like MPI and flexible grid-oriented models like Satin. JEL allows programming models like Satin to implement both malleability and fault tolerance. With JEL as a common layer for resource tracking, the development of programming models is simplified considerably. In the Ibis project, we developed a number of programming models using JEL, and we continue to add models regularly.

JEL can be used on a number of environments, ranging from clusters to highly dynamic P2P environments. We described several implementations of JEL, including a centralized implementation that can be combined with decentralized dissemination techniques, resulting in high performance, yet with low resource usage at the central server. Furthermore, we described several dissemination techniques that can be used with JEL. These include a broadcast tree and gossiping-based techniques. In addition, we showed that JEL can be implemented in a fully distributed manner, efficiently supporting flexible programming models like Satin, and increasing fault tolerance.

There is no single resource tracking model implementation that serves all purposes perfectly. Depending on the circumstances and requirements of the programming model and application a different implementation is appropriate. In a reliable cluster environment, a centralized implementation performs best. If applications are run on low bandwidth networks, the broadcast tree dissemination technique has the benefit of using very little bandwidth. In a hostile environment, such as desktop grids or P2P systems, a fully distributed implementation is robust against failures. JEL explicitly supports different algorithms and implementations, making it applicable in a large number of environments.

We evaluated JEL in a number of real-world scenarios. The scenarios include starting 2000 instances of an application, wide-area tests with new machines joining, and resources failing, and running an application on a worldwide system, including grids, P2P systems, and cloud computing resources. In addition to these experiments, we have won a number of international competitions, showing the suitability of JEL for real-world applications.

The future work consists of implementing additional programming models using JEL, such as a distributed hash table (DHT), and redesigning our implementation of the Satin divide-and-conquer model to explicitly support low connectivity environments. In addition, we plan to implement a fully distributed version of JEL that supports reliable joins and leaves and uniform elections. One way of implementing this would be using Lamport clocks [16] and a distributed election algorithm [17].

ACKNOWLEDGEMENTS

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been supported by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment).

DOI: 10.1002/cpe

We kindly thank Ceriel Jacobs, Kees Verstoep, Roelof Kemp, Nick Palmer and Kees van Reeuwijk for all their help. We would also like to thank the people of the InTrigger grid (Japan) for access to their system. We also like to thank the anonymous reviewers for their insightful and constructive comments.

REFERENCES

1. Foster I, Kesselman C, Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 2001; **15**(3):200–222.
2. Drost N, van Nieuwpoort RV, Bal H. Simple locality-aware co-allocation in peer-to-peer supercomputing. *CCGRID '06*: *Proceedings of the Sixth IEEE International Symposium on ClusterComputing and the Grid*. IEEE Computer Society: Washington, DC, U.S.A., 2006; 14.
3. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: The condor experience: Research articles. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(2–4):323–356.
4. Amazon ec2 website. http://aws.amazon.com/ec2 [1 March 2010].
5. Nieuwpoort R, Wrzesinska G, Jacobs CJ, Bal HE. Satin: A high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 2010; **32**(3):1–39.
6. MPI forum website. http://www.mpi-forum.org/ [1 March 2010].
7. Nieuwpoort R, Maassen J, Wrzesi«ska G, Hofman RFH, Jacobs CJ, Kielmann T, Bal HE. Ibis: A flexible and efficient java-based grid programming environment: Research articles. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(7–8):1079–1107.
8. Postel J. Transmission Control Protocol. *RFC 793* (*Standard*), September 1981. Updated by RFCs 1122, 3168.
9. Bornemann M, van Nieuwpoort RV, Kielmann T. MPJ/Ibis: A flexible and efficient message passing platform for Java. *Proceedings of PVM/MPI'05*, Sorrento, Italy, September 2005.
10. Bal HE, Drost N, Kemp R, Maassen J, van Nieuwpoort RV, van Reeuwijk C, Seinstra FJ. Ibis: Real-world problem solving using real-world grids. *IPDPS '09*: *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society: Washington, DC, U.S.A., 2009; 1–8.
11. Waldo J. Remote procedure calls and java remote method invocation. *IEEE Concurrency* 1998; **6**(3):5–7.
12. Maassen J. Method invocation based communication models for parallel programming in Java. *PhD Thesis*, Vrije Universiteit, Amsterdam, The Netherlands, June 2003.
13. Goux J-P, Kulkarni S, Yoder M, Linderoth J. An enabling framework for master-worker applications on the computational grid. *HPDC '00*: *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society: Washington, DC, U.S.A., 2000; 43.
14. Taura K, Kaneda K, Endo T, Yonezawa A. Phoenix: A parallel programming model for accommodating dynamically joining/leaving resources. *PPoPP '03*: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM: New York, NY, U.S.A., 2003; 216–229.
15. Fagg GE, Gabriel E, Bosilca G, Angskun T, Chen Z, Pjesivac-Grbovic J, London K, Dongarra JJ. Extending the MPI specification for process fault tolerance on high performance computing systems. *Proceedings of ICS'04*, Saint-Malo, France, June 2004.
16. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
17. Gupta I, Renesse Rv, Birman KP. A probabilistically correct leader election protocol for large groups. *DISC '00*: *Proceedings of the 14th International Conference on Distributed Computing*. Springer: London, U.K., 2000; 89–103.
18. Kielmann T, Hofman RFH, Bal HE, Plaat A, Bhoedjang RAF. Magpie: Mpi's collective communication operations for clustered wide area systems. *PPoPP '99*: *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM: New York, NY, U.S.A., 1999; 131–140.
19. Drost N, Ogston E, van Nieuwpoort RV, Bal HE. Arrg: Real-world gossiping. *HPDC '07*: *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. ACM: New York, NY, U.S.A., 2007; 147–158.
20. Leach P, Mealling M, Salz R. A Universally Unique IDentifier (UUID) URN Namespace. *RFC 4122* (*Proposed Standard*), July 2005.
21. Jelasity M, Guerraoui R, Kermarrec A-M, van Steen M. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. *Middleware '04*: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. Springer: New York, NY, U.S.A., 2004; 79–98.
22. Smith T, Watherman M. Identification of common molecular subsequences. *Journal of Molecular Biology* 1981; **147**:195–197.
23. Intrigger website. http://www.intrigger.jp [1 March 2010].
24. Maassen J, Bal HE. Smartsockets solving the connectivity problems in grid computing. *HPDC '07*: *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. ACM: New York, NY, U.S.A., 2007; 1–10.
25. Beck M, Dongarra JJ, Fagg GE, Geist GA, Gray P, Kohl J, Migliardi M, Moore K, Moore T, Papadopoulous P, Scott SL, Sunderam V. Harness: A next generation distributed virtual machine. *Future Generation Computer Systems* 1999; **15**(5–6):571–582.

DOI: 10.1002/cpe

26. YarKhan A, Dongarra J, Seymour K. Gridsolve: The evolution of network enabled solver. *Proceedings of IFIP WoCo9*, Prescott, AZ, U.S.A., July 2006.
27. Babaoğlu O, Bartoli A, Dini G. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computer* 1997; **46**(6):642–658.
28. Eugster PT, Felber PA, Guerraoui R, Kermarrec A-M. The many faces of publish/subscribe. *ACM Computing Surveys* 2003; **35**(2):114–131.
29. Czajkowski K, Kesselman C, Fitzgerald S, Foster I. Grid information services for distributed resource sharing. *International Symposium on High-Performance Distributed Computing*, Redondo Beach, CA, U.S.A., 2001.

DOI: 10.1002/cpe