Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Design of an Infrastructure for Highly Available and Scalable Grid Services
D3.2.1

Due date of deliverable: November 30^{th}, 2006
Actual submission date: December 21^{st}, 2006

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2

Responsible institution: VUA
Editor & and editor's address: Guillaume Pierre
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1081a
1081HV Amsterdam
The Netherlands

Version 1.4 / Last edited by Guillaume Pierre / December 21^{st}, 2006

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	$\surd$
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	2006/10/16	Guillaume Pierre	VUA	first draft (including contributions from ULM)
0.1	2006/10/30	Paolo Costa	VUA	Additional VU contribution
0.2	2006/10/31	Jan Stender/Bjoern Kolbeck	ZIB	Publish/subscribe section
0.3	2006/10/31	Michele Albano, Diego Puppin	CNR	Resource discovery section
0.4	2006/11/06	Guillaume Pierre	VUA	Many minor wording improvements
0.5	2006/11/07	Jörg Domaschka	ULM	Additional ULM contributions
0.6	2006/11/07	Björn Kolbeck	ZIB	Additional ZIB contribution
1.0	2006/11/09	Guillaume Pierre	VUA	Document wrap-up for internal review
1.1	2006/11/22	Guillaume Pierre	VUA	Updates to take internal reviews into account
1.2	2006/11/23	Jörg Domaschka	ULM	Minor updates following reviewer remarks
1.3	2006/11/27	Guillaume Pierre	VUA	Document wrap-up
1.4	2006/12/21	Guillaume Pierre	VUA	Changed document date

Contents

Executive summary

Workpackage 3.2 of the XtreemOS project aims at *providing an infrastructure that can support highly available and scalable grid services and applications*, such that these can be developed independently from underlying instances of the XtreemOS operating system. This document presents an initial design of the mechanisms we plan to build to allow service programmers to benefit from desirable system functionality and non-functional properties such as node management, application initialization, stable access points, high availability, publish-subscribe and resource monitoring:

Node Management This is used to discover nodes in the Grid that are capable of running a job.

Application initialization This is used to allow newly started applications to organize themselves and establish a structure between their respective nodes.

Distributed server This is used to provide a stable contact address to a group of nodes such that the application can always be contacted there, irrespective of changes in the membership of the application.

Virtual Node This is used to group several physical nodes into fault-tolerant virtual nodes such that node failures can be hidden to the applications.

Publish-subscribe This is used as an internal communication mechanism for many end-user applications and grid services, such as the XtreemFS shared file system.

Directory service for node monitoring This is used to store and retrieve structured information about the Grid. The prime application is to support a job and resources monitoring service for the Grid.

1 Introduction

1.1 Goals

Workpackage 3.2 of the XtreemOS project aims at *providing an infrastructure that can support highly available and scalable grid services and applications*, such that these can be developed independently from underlying instances of the XtreemOS operating system. We propose to develop mechanisms to allow service programmers to benefit from desirable system functionality and non-functional properties such as node management, application initialization, stable access points, high availability, publish-subscribe and resource monitoring.

We can perceive the XtreemOS grid as a huge heterogeneous collection of machines that users can allocate to execute their applications. The collection can be very big, in the order of at least 100,000 machines distributed worldwide. Such a scale is not unrealistic for the near future if we consider that for example Google has already reached this kind of scale [10]. How this collection has been formed is, for now, not relevant. However, as an example, one can imagine that various partners of the XtreemOS consortium have each contributed one or more (clusters of) computers in a way similar to what is now done in PlanetLab [1].

Applications running on XtreemOS may require in the order of hundreds to thousands of nodes each, depending on their specific purposes. Applications may be standalone, in which case they barely have any interaction with the external world except with the user that launched them. They can also be *services*, in which case they present an interface to the outside world so that they can be invoked.

Given the scale that we consider, it is expected that no centralized solutions will be feasible. Also, the infrastructure used to execute a given application will likely change over time as some machines may fail or be allocated to other tasks, and/or others be dynamically added during execution.

1.2 Terminology

We define a *service* as a group of processes that provides a standardized interface to the outside world. External programs can query the service by sending network messages to the service's interface. Services can be used to implement system-level functionality such as global resource discovery and allocation across the grid or controlling the execution of jobs. Services can also implement application-level functionality, for example a publicly-available Web service that exploits the resources of the Grid to provide guaranteed response times [12]. Services may be composed of other services, which may belong to different users and organizations.

From the point of view of its clients, a service is a simple interface that they can query; from the point of view of the operating system, a service is just another type of application that it needs to execute.

1.3 Document organization

This document describes a detailed design of the mechanisms that WP3.2 will build. In Section 2, we outline the constituents elements of our approach which will be detailed in the following chapters. In Section 3 we discuss our proposal to discover nodes likely to be allocated to applications. In Sections 4, 5, 6, 7 and 8 we discuss the basic tools that we plan to build to support the development of Grid services. In Section 9 we discuss the relationships between WP3.2 and other workpackages within XtreemOS. Finally, in Section 10 we present our work plan to realize the previously presented tasks.

2 Design overview

2.1 Use-case scenario

Let us consider a simple grid application that consists of a single program and some data that are to be distributed and replicated across multiple machines. Assume that this application implements a service that can be accessed by other applications. To deploy this application, the developer would provide a specification of the minimal requirements that a computer should meet. To keep matters simple, we assume that these requirements can be formulated as a set of ⟨attribute,value⟩ pairs, in which the attribute refers to statically available resources such as disk size, memory size, CPU type, available libraries, etc. In addition, the number of required nodes is specified as well.

The application's description, along with its system requirements are submitted to one of many entry points of the XtreemOS infrastructure for acceptance. Practically, this means that some authority will check whether the application can, in principle, be hosted by the infrastructure. If so, then the requested nodes will be allocated from the available resources within the relevant Virtual organization, and the application will be uploaded and executed at each of these nodes. To allow the application to initialize itself, the system will materialize the collection of allocated nodes into a peer-to-peer overlay and maintain it as nodes join or leave the application. Each instance of the application can query this overlay for a random subset of addresses where other instances can be contacted.

What happens after this point depends entirely on the application.

The application will keep on running until its instance processes terminate, or it is explicitly interrupted by its user or an authorized administrator.

Application initialization

Some applications may start their computation based entirely on their local information without attempting to communicate among different instances. However, we will provide libraries that the applications can use to address common needs. For example, there will be multiple libraries capable of structuring the nodes of an application into different patterns (e.g., organize nodes into a DHT, organize them into an N-dimensional matrix, rank them so that they can be addressed using MPI, etc.).

Virtual nodes

Given the size of the system, node failures cannot be ignored. Certain applications may be capable to operate with slightly different number of resources than requested. Alternatively, we will provide a library that allows to define *virtual nodes*, that is fault-tolerant groups of nodes capable of taking over each other's tasks.

Distributed servers

Certain applications need to export an interface to the outside world that client can access to issue their requests. In such cases, the number of nodes involved and the dynamicity can make it difficult to advertise a single entry point to the clients. We will provide tools to build *distributed servers* capable of hiding the internal organization of the application and present a single stable address that clients can always access to reach the application.

Others

We will develop several other types of facilities to help applications solve classical problems. Among them will be facilities to support *Publish/Subscribe* communication, and a scalable *directory service* primarily intended for monitoring and node failure detection.

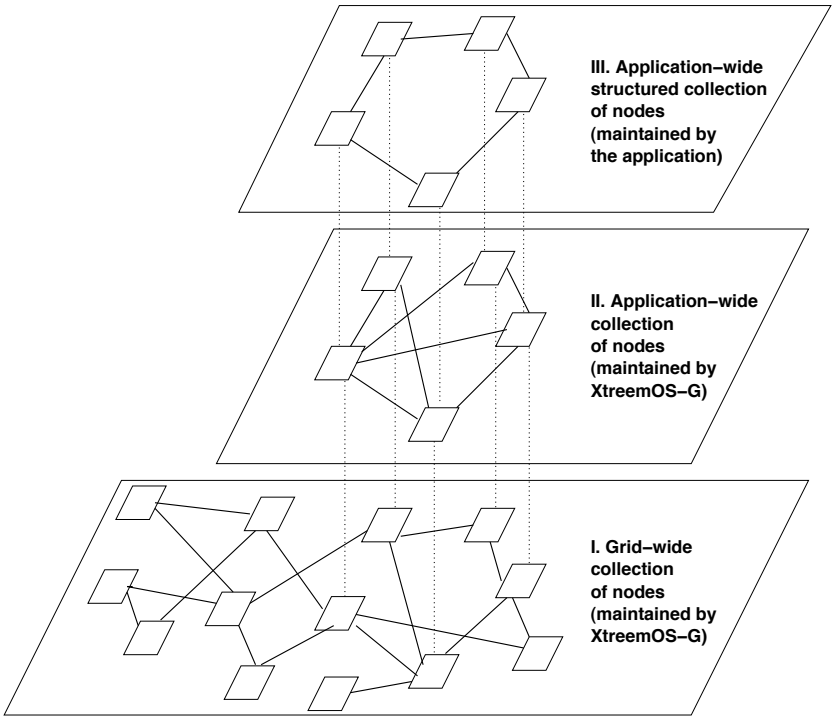

Figure 1: The three different collections supported by XtreemOS.

2.2 General architecture

The system to be built in WP3.2 can be seen as being made of a number of collections of nodes. Each collection is materialized as some form of overlay that keeps the nodes in the collection connected to each other. Importantly, because of the size and the dynamicity of the system we should build our solutions such that no node in a collection needs to have access to the full list of nodes taking part in it at a given moment.

Certain node collections are maintained by the node allocation system known as XtreemOS-G. Some others are maintained by the applications themselves, and possibly implemented in libraries provided to the application developers (see Figure 1).

2.2.1 Collections

The Grid is materialized by a *grid-wide collection* that comprises all nodes taking part in it. Being part of this collection is equivalent to belonging to the grid. Once an application has been allocated nodes to run on, an *application-wide unstructured collection* is created to keep all nodes taking part in the application connected. This collection is made visible to the application, and can be queried

for a random subset of other nodes taking part in the same collection. This collection is maintained over time by XtreemOS-G as a background process, so when nodes join or leave the application they simultaneously join or leave this collection. More details on XtreemOS-G can be found in Section 3.

Each application can decide to instantiate one or more *application-wide structured collections*. Such collections allow the nodes of an application to organize themselves in a given structure. For example, rather than requesting for random subsets of other nodes in the application, one may organize nodes in an N-dimensional matrix, a farmer-worker structure, a Chord ring, a ranked list likely to be used with MPI, etc. Specialized collections are implemented in libraries organized as a toolbox. They rely on information provided by the application-wide unstructured collection for their own management.

The last kind of collection is a *virtual node*. Several nodes taking part in a given application can organize themselves into a virtual node, i.e., a fault-tolerant group of nodes. Mode details on virtual nodes can be found in Section 6.

2.2.2 The toolbox

The toolbox contains a number of useful abstractions in the form of one or more libraries. It is intended to be used by end-user application programmers but also by other system-level service developers, such as those built by WP3.3 and WP3.4. The toolbox will contain the following tools (note that this is by no way an exhaustive list):

- *Facilities to construct structured collections*: we will provide several instances of such facilities specialized for building different types of structures. These facilities are discussed in Section 4.

- *Distributed servers*: nodes taking part in an application can request to be organized as a distributed server. A distributed server is a collection of nodes with a single address that clients can contact to reach the application. This address remains stable, even in the case of nodes joining or leaving the application. A possible implementation of distributed servers is discussed in Section 5.

- *Virtual nodes*: a group of nodes taking part in an application can request to be organized as a virtual node. A virtual node is a fault-tolerant group where each member can take over the task of the others in case of failure. Several types of virtual nodes may be provided, based on active replication, passive replication, and checkpoint/restart mechanisms provided by the XtreemOS operating system. Virtual nodes are discussed further in Section 6.

- *Publish/subscribe*: a common form of communication between large number of nodes taking part in a given application is publish-subscribe. We will provide a fully decentralized pub/sub communication system that applications can use for their own purpose. Note that the implementation of structured collections may require the availability of a basic pub/sub mechanism, too. Whether both pub/sub systems can be developed as one generic entity or if two different ones are preferable is left for future decision. Publish-subscribe services are discussed in Section 7.

- *Node monitoring and failure detection*: in a large-scale environment, it becomes increasingly difficult to monitor resources whose utilization may change dynamically. To address this issue, we will provide a node monitoring service based on a directory service. We expect that a single implementation of the directory service will not suffice and that, depending on the dynamicity of the published attributes, different system architectures will be used automatically. Node monitoring facilities are discussed in Section 8.

3 Node management

Node management plays a crucial role in the XtreemOS system, as it should be able:

1. to connect the XtreemOS nodes in the *grid-wide collection*;

2. to provide a mechanism to discover and allocate the required set of nodes, according to the application specification;

3. to create and maintain an overlay network connecting the nodes allocated to the application (*application-wide unstructured collection*). This overlay network can be used directly by the application or be the basis to build an *application-wide structured collection*, using one of the tools described in Section 2.

There are obviously different approaches to address these goals. In a fully centralized approach, a *node management entity* can keep track of all nodes in the infrastructure, their current allocations, and their currently available resources. Such an approach has obvious scalability problems. Given the requirements of our envisioned scenario, we should instead address these problems using fully decentralized solutions, enabling scaling up to thousands of nodes, which may join and leave in a very unpredictable way.

A radically different approach is to let the nodes self-organize into a collection that meets the specifications. In essence, this means that the node management

entity is fully decentralized across the nodes that are part of the grid infrastructure. Our systems will rely on existing epidemic techniques that have shown the potential to build very large-scale self-organizing systems [18, 19]. We plan to extend these protocols to address the peculiarities of the XtreemOS system, by dynamically arranging nodes in different collections according to the application specifications.

In the following, we briefly sketch our approach to node management and show how we can exploit it to cope with failures and dynamic topologies.

3.1 Grid-wide collection

As mentioned above, the first goal is to maintain a strong connectivity among nodes belonging to the XtreemOS network. This can be achieved by means of the CYCLON protocol [18]. This protocol provides a robust network connectivity typical of a random graph and exhibits fast convergence properties.

While Cyclon provides guarantees about the connectivity of the system, it does not support any other functionality by itself. We should therefore devise an additional mechanism to arrange nodes according to a topology which simplifies the process of node discovery. This, again, can be implemented on top of epidemic protocols as shown in literature [7, 19]. Using these techniques, combined with gossip-based decentralized aggregation methods [8], nodes can self-organize into semantic clusters, defined by commonality criteria.

When an application issues a request for a given amount of nodes with specific characteristics, the query will be routed to the semantic cluster which best approximates its requirements. This cluster can then be passed to the resource allocation mechanisms built in WP3.3 for further selection.

3.2 Application-wide collection

Once the nodes requested by the application have been discovered and allocated, these will be organized in a random graph topology, which can be used by the application to start its tasks. However, very few (if any) applications will be able to directly exploit such a random topology. Alternatively, as exposed in Section 2.2.1, the application may leverage additional libraries to shape this node collection into different structured topologies.

However, given the high degree of dynamicity characterizing the scenario we target, our system should be able to maintain these collections even in presence of unannounced disconnections, and possibly replace failed nodes with new ones. To this end, our proposal is to devise a cross-layer communication between the application-wide collection and the aforementioned grid-wide collection. The latter thus becomes aware of the need of the former and as soon as new nodes become

available, these are added to the application collection which can resort to them when the overall number of allocated nodes falls below the desired threshold.

3.3 Application deployment

Another service that should be provided by the XtreemOS system concerns the deployment of the application code. A very trivial solution would be to let every node download the binaries from an external repository (e.g., using HTTP or GridFTP), whose address is stored in the query. Although in principle this solution could work, it may become unsuitable for large application collections which would saturate the repository bandwidth soon. To circumvent this issue, we could for example rely on the grid file system developed by WP3.4 or by a shared download mechanism like BitTorrent [3] to spread the network load uniformly among nodes.

3.4 Planned demonstration applications

To show the effectiveness of our approach, we will develop a test-bed running on an existing grid infrastructure like PlanetLab [11] or Grid5000 [2]. A possible demonstration of our technology is to select a subset of hosts to run one or more applications, according to the techniques introduced above. During the application execution, we will artificially shut down some nodes and our system should be able to replace them using available nodes in a transparent way with respect to the application.

Note that we can also imagine emulating a very large-scale grid on top of limited resources by running multiple virtual hosts on the same physical host.

In a second phase, once our systems have been reasonable tested, we will apply them at deploying selected applications from WP4.2.

4 Application initialization

We may not expect that the initial composition of a distributed server remains the same during the lifetime of its hosted grid application. The grid application may grow or shrink dynamically, causing changes in the required resources. Likewise, we cannot expect that allocated nodes will continue to be available, requiring that nodes are replaced when they leave the cluster (voluntarily or due to a failure). For these reasons, the composition of the distributed server needs to be monitored and kept up-to-requirements dynamically.

Nonetheless, thanks to the epidemic nature of our protocol, our system can easily cope with these failures. Indeed, application-wide network connectivity

can be achieved by having application nodes periodically exchange the list of neighbors. In addition, thanks to the underlining gossip protocols, as soon as new nodes become available (either because they join the XtreemOS network or because they have been deallocated) they will be discovered by nearby nodes (i.e., nodes sharing similar attribute values). This way, if nodes belonging to an application collection should detect new available nodes, they could be easily added to the application collection and become part of the application overlay network.

The XtreemOS toolbox consists of a collection of libraries offering various ways to organize the processes that constitute a distributed application. We refer to the process executing application code on a specific node as a *constituent process* of that application. Together, the constituent processes of an application form a *distributed process* encapsulating the application analogous to the encapsulation of a program by a process on single-processor operating systems.

4.1 Application initialization

Jelasity and Babaoglu [7], but also Voulgaris and van Steen [19] have shown how structured overlays can be constructed using relatively simple epidemic protocols in unstructured networks. Using the information provided by the underlining protocol described in Section 3, we can adopt a similar approach in which the higher layer requests the lower layer for randomly selected peers, but maintains a list of only those peers that satisfy a specific distance metric. Such a metric may be related to semantic equivalence, expressed, for example, as the number of files that the requesting and selected peer have in common. Another example is when peers are ranked according to the distance between IDs, which is essentially exactly what happens in structured overlays.

The two layers need not be strictly separated, and, in fact, much higher convergence speeds are obtained if they are not [19].

4.1.1 Example: Chord

As an example, consider the construction of a Chord ring [7]. We assume that every process is assigned a randomly chosen unique m-bit identifier from the set $[0, 2^m - 1]$. Every process is initially assigned a random selection of nodes, for example through a lightweight membership protocol as described in [6]. The distance between two nodes with ID a and b, respectively, is simply defined by their difference in IDs: $d(a, b) = \min\{|a - b|, N - |a - b|\}$.

Each constituent process with ID p maintains a list of nearest nodes in order to construct the finger table. In particular, for each $1 \leq j \leq m$, a process stores the peer with the smallest identifier q satisfying

$$p + 2^{j-1} \bmod 2^m \leq q < p + 2^j \bmod 2^m$$

Note that this approach need not guarantee that each finger table entry actually stores the so-called successor, as required by Chord. The successor of key k is defined as the node with the smallest identifier $id \geq k$. However, the sketched construction of a Chord-like ring establishes properties similar to that of Chord.

We anticipate that variations of well-known structured overlays such as Chord are supported as well. For example, a more efficient version of Chord, described in [17] does not maintain finger table entries in the key space, but in the node space. Such a variation can be constructed on top of the basic overlay, but perhaps not by means of another higher-level *epidemic* protocol.

Typically, the Chord toolbox not only organizes the processes into a ring, but also provides the functions for looking up keys, configuring replication degrees, and so on.

4.1.2 Example: ordered IDs

As another example, consider the situation in which we want to assign each process a unique identifier $id \in \{0, \ldots, N-1\}$ where N is the number of constituent processes. Such a numbering may be needed to organize the processes into a farmer/worker configuration in which process 0 is the farmer and communicates with the $N-2$ workers.

In this example, we need to automatically rank each process. To simplify matters, let us assume that N is known, but that each process has been assigned a randomly selected unique m-bit identifier, with $N \leq 2^m$. A simple, yet somewhat costly algorithm is to disseminate process IDs to all nodes using epidemic-based gossiping. Each process maintains an ordered list of all process IDs it gradually learns (along perhaps with the associated network-level addresses), and continuously recomputes its own rank in this list. Obviously, this computation converges to the proper rank of a node, and terminates when each process will have learned about every other process.

4.2 Planned demonstration applications

We plan to demonstrate the application initialization facilities by developing applications that require specific organization (N-dimensional mesh, MPI-like organization, etc.). The fact that nodes participating in the application do not know the entire group membership will be hidden to applications, so that each application can use its own preferred addressing scheme for communication.

5 Distributed server

As discussed in Section 1.2, a service is defined as a group of processes which publish a standardized interface and can receive invocations by external clients. This means, of course, that clients need to know *where* invocations should be issued. For simple services running on a single stable machine this is relatively easy: one can for example hardcode the service address in the stubs to be instantiated at the clients.

Binding clients to service instances becomes much more difficult when the service is implemented as a group of processes running on a varying collection of machines. The Grid services that we envision may need to adjust the collection of servers on which they run for many reasons, such as maintaining a guaranteed response time regardless of variations of the load addressed to them, or to replace failing machines with fresh ones. In the context of frequent changes in the membership of the service, binding clients to a suitable service instance becomes much more difficult. As we discussed in [16], the required feature here is *anycast*, which allows to define groups of processes and allows clients to address any one of them. However, for anycast to be useful in this case, it needs to satisfy a number of non-functional requirements: first, the service should retain control on which client request should be treated by which service instance. This is most important in Grid services such as a distributed file system, where for performance reasons clients must be directed to the closest possible instance of the service. Second, the anycast implementation must support frequent reconfigurations of the set of service instances, as this set may change at any time. This includes cases where a service instance fails or gracefully leaves the system while it still has long-standing connections open with a number of clients. Finally, deploying the anycast implementation should be easy and require no special network administrator privilege. None of the traditional techniques based on centralized frontends, client-side software, DHTs, routing-based anycast and DNS redirection can satisfy all these requirements at the same time.

Instead, we propose a solution called *versatile anycast* [15]. Versatile anycast relies on the Mobile IPv6 protocol [9] to represent a group of processes running on a varying collection of machines to be recognized by its clients as having a single stable IPv6 address. Changes in the composition of the set of service instances are totally transparent to the client application. However, the service can control on a per-client basis which service instance should handle incoming requests. Unlike other implementations of the anycast functionality, we have shown that this solution fulfill all the previously mentioned functional and non-functional requirements.

WP3.2 will deliver libraries that can be used by service programmers to use versatile computing functionalities. In essence, this will allow service instances

to hand-off client connections between themselves, thereby controlling which service instance should handle which client requests.

5.1 Planned demonstration applications

A possible demonstration application is to implement a replicated Web server. Multiple service instances contain copies of the same documents, and are organized as a distributed server. Clients keep on addressing their requests to the same versatile anycast address, and receive responses despite frequent changes in the composition of the distributed server.

We then plan to work in cooperation with partners from WP4.2 to apply the distributed server techniques to applications such as Wissenheim, SPECweb2005 and Web Application Server.

6 Virtual node

Normally application constituents are executed on a single node. Sometimes, however, this may be too dangerous, because a certain constituent is a critical part of the entire application and its failure cannot be tolerated. In another scenario one may think to have a service for which the number of read-only operations outperforms the number of operations that modify the service's state, e.g. a web server. In both cases having a constituent on a single node can lead to severe drawbacks, namely the breakdown of the entire application in the former case and a performance bottleneck in the latter.

We consider *replication* as a reasonable approach to solve both problems, as it may provide higher availability and resilience against unrelated failures of individual nodes and is also able to distribute read operations to a higher number of nodes. A group of constituents running replicated is called a virtual node.

At first view a virtual node resembles a distributed server. Looking closer, however, there are three significant differences:

- *Accessors*: Distributed servers are built for all kinds of clients. Virtual nodes serve only nodes running XtreemOS and being part of the grid as they only serve nodes that are within a distributed server.

- *Transparency*: Virtual nodes do not hide the fact that they are distributed. Clients can benefit from this knowledge, if they want, but do not have to.

- *Scale*: The scale of a virtual node is at the very most up to a few dozen real nodes, whereas a distributed server can easily consist up to 100,000 nodes. Larger scales for virtual nodes will barely be feasible due to the intrinsic

costs of total order messaging that is an evident precondition for replication systems.

6.1 Construction

For constructing a virtual node we can use the same process as for constructing a distributed server [15]. In essence, a virtual node is an abstraction of a real node, but one that provides high availability. Similar, there is a need for some specifications upon which an algorithm can decide which nodes to choose. However this algorithm will not operate on the entire set of nodes, but only on the nodes already allocated for the application. From the grid and registration levels' point of view the collection of nodes built for a virtual node is just one more application dependent overlay that also uses the mechanisms provided by lower levels to request new nodes or to be informed about nodes failures. There will be another virtual node overlay on top of the first one representing the communication and distribution hierarchy and infrastructure of the virtual node. That is, if active replication with total message ordering is used, the topology of this overlay will probably be fully meshed and will guarantee total message ordering, whereas in an extreme case of passive replication, it might be a star and with no or only weak consistency guarantees.

6.2 Replication

When talking about replication we think about replicating distributed objects mainly for fault-tolerance [14], but also for performance reasons. This is quite different from database replication as the semantics of an operation are mostly unknown. In principle there are two main strategies how to replicate: active and passive, but of course also hybrid setups can be imagined. All three of them can be combined with a number of different consistency and communication strategies. Due to this diversity we will provide support for active replication first. Active replication, however, makes evident the need for determinism in all participating replicas. This demand makes multi-threading a non-trivial task, as all schedulers have to take the decisions in the same logical order [13, 4]. Instead of using only single-threaded execution as most systems nowadays do, we will also provide multi-threading support.

In a next step we will extend this architecture to support passive replication. Finally, we will investigate how the system can be changed so that support for additional communication patterns and consistency requirements can be added easily and exchanges can happen dynamically at runtime to adapt to a changing environment. The different approaches will be integrated in the toolbox.

Different replication modes actually have different limitations in their particular programming model. To allow strategy exchange there is either need for a common programming model enabling all kinds of replication or a code checker that inspects the source code and decides which strategies can be used with the code.

6.3 Access point

Like the distributed server, a virtual node must provide an access point that other server nodes can use for accessing it. As mentioned above this does not have to happen in a completely transparent way. However, as replication normally is a non-functional requirement, transparency might be a useful feature from an application programmer's point of view, because the application could also be run non-replicated. As node-to-node communication is thought to happen by the application level communication structure presented in Section 2, accessing the virtual node is equal to sending a message on this overlay. Thus, the way the nodes participating in a virtual node are integrated in the application-wide overlay and the communication pattern used are crucial for accessing a sufficiently high number of nodes and for guaranteeing accessibility.

Regarding the non-redundant star-topology mentioned above, it would suffice to integrate the center node, as it marks the entry point to the virtual node. In the fully meshed scenario all of the nodes can contribute in the application overlay (but do not have to).

6.4 Planned demonstration applications

We plan to demonstrate the functionality of a virtual node by providing a simple multi-threaded, fault-tolerant service. Furthermore we will provide techniques that allow clients to access this service transparently in face of location, migration, relocation, concurrency, and replication without the need for major changes in client code or no changes at all.

Furthermore we plan to investigate if and how virtual nodes can be applied to the applications in work package 4.2, but also to other system wide services, like e.g. the file system's meta data server. Therefor we will of course depend on the contributions and cooperation of the particular supplier.

7 Publish-subscribe

The goal of this task is to offer a scalable and fault-tolerant publish/subscribe system for asynchronous and loose coupling of applications in a grid. The ser-

vice will be used by XtreemOS components for monitoring, notification of server crashes, as well as for persistent queries in the XtreemFS and job events for execution management. In addition, the system can also be used by user applications for different purposes.

Given that several other workpackages and tasks within our own workpackage rely on the quick availability of a working publish/subscribe system, we will organize our work in two steps. First, we will build a solid implementation that is highly scalable but has relatively limited features, and can be used as a base by other groups. We will then explore advanced extensions, to support extra features such as range-based subscriptions, support for generalized multicast and atomic updates.

We decided to initially implement a topic-based system, since this type is easier to scale than content-based publish/subscribe systems. In this section we describe the ideas for a first prototype which will be implemented quickly to offer basic services to other workpackages as soon as possible. Our prototype will be extended with features resulting from research outlined in section 7.5.

7.1 General definitions

Event An event is a piece of information which is delivered asynchronously.

Topic In a topic-based pub/sub system, a topic describes a class of events for which a client can advertise interest by creating a subscription.

Subscription A subscription is used by a client to advertise interest in events pertaining to a certain topic. It consists of a communication endpoint and some representation of a topic. In order to allow for garbage collection of outdated subscriptions, it might be reasonable to associate a subscription with a lease time.

7.2 Architecture

We use a two tier architecture for our system. Clients (publishers and subscribers) register with a pub/sub server. The servers are organized in a structured overlay network. Figure 7.2 illustrates how a topic-based publish/subscribe system can be built on top of a structured overlay network. Information is stored using topics as keys and lists of communication endpoints as values.

Moreover, each node in the overlay network maintains a table of connected clients with active subscriptions. Adding a new subscription to a topic A to the system (step 1) first causes the corresponding server to update its table of local subscribers (step 2), and then calls the put() function of the overlay network, so as

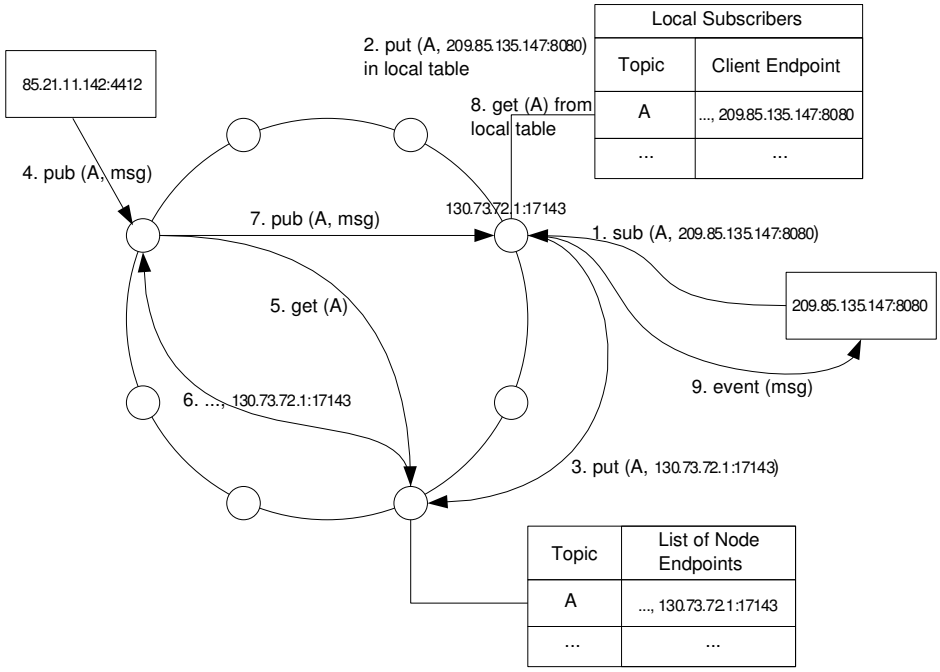

Figure 2: Example for a subscribe and event notification using an overlay network

to add its communication endpoint to a globally accessible list of communication endpoints associated with A (step 3). When an event associated with topic A is published (step 4), a lookup of all servers with active subscriptions to A is performed by invoking the get() function of the overlay network (steps 5 and 6), and the notification is forwarded to this list of servers (step 7). Finally, the event is delivered to the set of local clients by each of the servers that receive the notification, with the aid of their local subscriber tables (steps 8 and 9).

Using an overlay network in order to construct a mapping from topics to servers implicitly disperses the responsibility for subscriptions, with the aim of providing scalability. We believe that there will be more clients than servers, as well as different popularities of different topics.

Another problem lies in notifying subscribed servers. A simple broadcast scheme will become a bottleneck once the number of servers grows. One approach lies in using gossip based mechanisms to build multicast trees. Since we are using an overlay network, it could be very promising to explore how the over-

lay itself can be used to construct efficient multicast mechanisms on top of it. This has the tremendous advantage that only one network is used for routing events and subscriptions.

7.3 Interface

Clients establish a TCP connection to the server which is used by the client to subscribe and to cancel subscriptions. The server uses the same connection to send events to the client. If the connection is closed or broken the subscription ends. A simple, textual protocol will be used for communication to achieve platform independence. Event payload however, can be arbitrary binary data. These samples give a first idea of the protocol we will probably use.

- Sample subscription request:

```
SUBSCRIBE "topic name" \r\n
```

- Sample unsubscription request:

```
CANCEL "topic name" \r\n
```

- Sample publish request:

```
PUBLISH "topic name" \r\n
Payload-length: 128\r\n
<128 byte of binary data>\r\n
```

- Sample event notification:

```
EVENT "topic name" hashID\r\n
Payload-length: 128\r\n
<128 byte of binary data>\r\n
```

7.4 Implementation

The system will be built on top of an existing P2P substrate. Available implementations will be evaluated according to their license, active development, usability for production environments and functionality. In a first prototype a simple inter-server notification scheme will be implemented. In the following implementation this protocol will be enhanced, e.g by using more sophisticated multicast algorithms.

To bootstrap the P2P substrate we require that servers register with the directory service. This information does not need to be up-to-date all the time. However, a sufficient number of available nodes must be registered.

7.5 Longer-term prospects

The presented system is intended as a starting point. On the one hand, we quickly need a simple but working prototype for other XtreemOS packages relying on the pub/sub system. On the other hand, our design is flexible enough to allow for a variety of research topics to be implemented later on. In this section we give some of the research aspects of the publish/subscribe system that we will investigate. Others may be added during the course of research.

- **Hierarchical topics**
  Hierarchical topics are a compromise between topic and content-based publish/subscribe. While systems using flat topics are easy to scale and offer excellent performance, they require application programmers to build workarounds for more sophisticated event structures. On the other hand, content-based systems offer powerful mechanisms for event filtering but do not scale (well). To implement a scalable publish/subscribe system with hierarchical topics we cannot take advantage of range queries. Instead a specific topic has to be matched onto ranges (subscriptions). We will investigate how Chord# [17] can be modified or extended to support this kind of ”range filters”.

- **Multicast in overlay networks**
  To efficiently disseminate events in the system we can take advantage of the structure of the overlay network. How this is done in detail heavily depends on the underlying overlay network used. As part of our research we will investigate different overlays, e.g. Pastry, Bamboo or Chord#, and the possible multicast schemes. [5] gives more details on broadcasts in overlay networks. Combining or applying traditional multicast mechanisms like gossiping to very dynamic systems like structured overlays becomes a research challenge.

- **Atomic updates**
  To support concurrent changes on subscriber lists we need transactions to combine a get() and subsequent put() request into an atomic update operation. This is not supported by current overlay network implementations. Research will be conducted in close co-operation with the SELFMAN EU project. In combination with ”range filters” for hierarchical topics this may be extended to transactions over several lists.

7.6 Planned demonstration applications

The main application that is planned consists of the XtreemFS system (WP 3.4) will make extensive use of the publish/subscribe system for various applications:

- notification of clients for persistent queries
- dissemination of OSD status information
- monitoring of services / servers

8 Directory service for node monitoring

The goal of this task is to build a scalable and efficient directory service that can be used to register static and dynamic attributes. We expect to use it as a basis for node monitoring in the XtreemOS platform, but it may also be relevant to other workpackages for tasks such as resource management and discovery.

In such environments, the set of shared resources is highly dynamic, with entities joining and leaving the system at unpredictable times. Moreover, features of shared resources can change as time goes by, so information about the features has to be updated in the system. It is therefore necessary to design the directory service with such constraints in mind. To this end, we expect to structure the system as a layer on top of the publish/subscribe system described in Section 7. A more detailed analysis and discussion between the concerned partners is necessary, though, to take a final decision on this design choice.

8.1 System requirements and architecture proposal

The directory service has a number of requirements. Among them, we focus on:

- *Scalability*: The set of resources and of participating entities must be able to get huge without affecting seriously the system.
- *Robustness*: The system must be able to cope with the join/leave events of the entities.

The system that satisfies the requirements must be efficient in searching for resources, even in the presence of a large number of entities, and it must be able to cope with join/leave events.

In our opinion, resources' features are characterized by their level of dynamicity, that depend both on the resource and on the feature at hand. We think that we should not look for a "one-size-fits-all" approach to the problem, and we should build a system that can use different mechanisms depending on the level of dynamicity of the search target.

We propose an alternative focus on the problem. We considered a number of use cases of the publish/discover system where the key attributes of a query are dynamic attributes that describe short termed features of the resources.

As an example, we can cite a query for disk space, that looks for a disk server that has a certain number of free megabytes. Or a query for computing power, that has three main attributes: a minimum number of CPUs, their minimum power and the maximum workload pending on them.

Depending on the kind of attribute that is inserted in a query, there are different mechanisms that are suitable for the job at hand. We will therefore implement multiple publish and search algorithms in the directory service such that the most appropriate one can be used depending on the dynamicity of the published/queried attributes.

8.2 Open problems

The described system suffers from a number of problems that have no standard solution and need to be solved by some research effort.

A first problem is the range query. A client's query can be, for example, for "a disk with 2 to 3 GBytes of free space". The literature on range queries on *Distributed Hash Tables* present some interesting ideas that can be used to cope with the problem. In particular, the Chord# system developed by ZIB may be a good substrate for such queries.

Another problem regards queries for set of resources. An example can be "1 to 6 disks, with a total free space of 10 GBytes". The system must be able to create subqueries for single resources and to aggregate the results for the entity which did the query.

Another open problem is the multi–attribute query, when the attributes are of different dynamicity. The query system will collect data using different mechanisms and then it will have to merge the collected data into an answer to the query.

8.3 Added value of the system

The prime planned application of the directory service is to support a monitoring system to store and query information about the runtime of the grid. Applications and Operating System services will access the information published by the monitoring system in real-time. Another use of the publish/discover system, is to broadcast messages to the entire peer-to-peer network.

8.4 Planned demonstration applications

The first release of the system will include the tools to create and manage a directory service with multiple publish and search algorithms depending on the dynamicity of the attributes. A daemon will publish information about the node on

which it will be installed, for example by exporting the information found in the `/proc` directory. One can then run global queries through the system to search nodes based on certain attributes.

9 Relationship with other work packages

WP 3.2 plays a crucial role within the XtreemOS project and must frequently interact with the other work packages. In this section we briefly describe the collaborations occurring with the rest of the project and propose a joint work plan.

9.1 Relationship with sub-project 2

Although most protocols we are going to develop do not require any special kernel support, we will exploit a number of facilities provided by the XtreemOS kernel. These relationships will result in a number of requirements that sub-project 2 should ideally support.

First of all, as specified in Section 5 we need Mobile IPv6 support to implement our distributed servers. Mobile IPv6 is implemented in all major operating systems, including Linux, so we expect no major problem there. In addition, we will also take benefit from the access control tools which will be used during the node discovery and allocation to select only trusted nodes and to prevent un-authorized users to run their code. Finally, the fault-tolerance mechanisms introduced in Section 6 will leverage the checkpoint/restart services implemented in kernel-space.

9.2 Relationship with WP3.1

Since one of our goals is to provide a set of libraries available to the applications (see Section 2.2.2), we need a tight interaction with WP3.1 and we will work together to define a suitable API for these services.

9.3 Relationship with WP3.3

WP3.3 represents the prime target user of our node management mechanisms. Indeed, after our protocol has identified a suitable set of nodes based on their static properties, it is expected that it will deliver this list to the allocation components developed by WP3.3, which will select only a fraction of them according to specific scheduling policies (e.g., load distribution, network latency, etc.). This means that our node management system can be seen as the implementation of the "ResourceDiscovery" component described in deliverable D3.3.1.

Beside this, we believe that other components we are going to develop could be of interest for WP 3.3 in the development of highly available and scalable services. For example, the node allocation components may use the fault tolerance support through virtual nodes and the directory service to retrieve node characteristics.

9.4 Relationship with WP3.4

Our directory service is central to the design of the file system to be developed in WP3.4. This, therefore, will require an active collaboration among the two work packages to jointly identify the common issues and the definition of the components we need to implement to support their service.

Also, the share file system developed by WP3.4 could become the prime communication medium for deploying Grid applications (see Section 3.3).

9.5 Relationship with WP3.5

Security represents a major concern of the XtreemOS project, but is not the prime focus of WP3.2. We consider that global security of the XtreemOS platform is the responsibility of WP3.5, but we will aim at securing the services that we will build as part of WP3.2. We expect to use primitives developed by WP3.5 for that. This will certainly require frequent interaction with WP3.5.

9.6 Relationship with sub-project 4

WP3.2 needs to closely collaborate with sub-project 4 to provide the services required by application developers. Some of these service (e.g., fault-tolerance) will be directly supported by our workpackage (although they may affect the overall performance) while some others (e.g., authentication, security and monitoring) will be implemented in other work packages. In the latter case we will build the necessary services, whose action will be mostly to mediate between existing system components inside the system.

Together with WP3.3, we will also do our best to provide efficient communication among nodes, although it is in the general case impossible to enforce strong guarantee in term of bandwidth and QoS between any group of nodes, even if they are allocated inside a given cluster.

10 Work plan

The table below illustrates the repartition of work between the members of the work package, as well as the planned efforts between months 1 and 18.

Partner	Tasks	Person-months
VUA	Node management, application bootstrapping, distributed server	45
ZIB	Publish-subscribe	27
CNR	Directory service and node monitoring	24
ULM	Virtual node, application bootstrapping	14

References

[1] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *First Symp. Networked Systems Design and Impl.*, pages 245–266, March 2004.

[2] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, November 2005.

[3] B. Cohen. Incentives Build Robustness in Bittorrent. In *First Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[4] Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Rüdiger Kapitza. Deterministic multithreading for java-based replicated objects. In *Proc. of the 18th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'06, Dalles, Texas, Nov 13-15, 2006)*, 2006.

[5] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured P2P networks. In *The 2nd International Workshop On Peer-To-Peer Systems (IPTPS'03)*, February 2003.

[6] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. In *Middleware 2004*, volume 3231 of *Lect. Notes Comp. Sc.*, pages 79–98, Berlin, October 2004. ACM/IFIP/USENIX, Springer-Verlag.

[7] Mark Jelasity and Ozalp Babaoglu. T-Man: Gossip-based Overlay Topology Management. In *Third Int'l Workshop Eng. Self-Organising App.*, June 2005.

[8] Mark Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based Aggregation in Large Dynamic Networks. *ACM Trans. Comp. Syst.*, 23(3):219–252, August 2005.

[9] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775, June 2004.

[10] John Markoff and Saul Hansell. Hiding in plain sight, Google seeks more power. *New York Times*, 14 June 2006.

[11] Larry Peterson, Andy Bavier, Marc Fiuczynski, Steve Muir, and Timothy Roscoe. Towards a Comprehensive PlanetLab Architecture. Technical Report PDN-05-030, PlanetLab Consortium, June 2005.

[12] Guillaume Pierre. Grid services for adaptive content delivery. In *Proceedings of the Workshop on the Use of P2P, GRID and Agents for the Development of Content Distribution Networks (UPGRADE-CDN)*, June 2006.

[13] Hans P. Reiser, Franz J. Hauck, Jörg Domaschka, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Consistent replication of multithreaded distributed objects. In *SRDS '06: Proceedings of the 25st IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, 2006.

[14] Hans P. Reiser, Rüdiger Kapitza, Jörg Domaschka, and Franz J. Hauck. Fault-tolerant replication based on fragmented objects. In *Proc. of the 6th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperabel Systems - DAIS 2006 (Bologna, Italy, June 14-16, 2006)*, 2006.

[15] Michał Szymaniak, Guillaume Pierre, Mariana Simons-Nikolova, and Maarten van Steen. Enabling service adaptability with versatile anycast. Submitted for publication, May 2006. `http://www.globule.org/publi/ESAVA_draft2006.html`.

[16] Michał Szymaniak, Guillaume Pierre, and Maarten van Steen. Versatile anycasting with mobile IPv6. In *Proceedings of the International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*, Pisa, Italy, October 2006.

[17] Thorsten Schütt and Florian Schintke and Alexander Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. In *Proceedings of the Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06)*, May 2006.

[18] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *J. Netw. & Syst. Mgt.*, 13(2):197–217, June 2005.

[19] S. Voulgaris and M. van Steen. Epidemic-style Management of Semantic Overlays for Content-Based Searching. In *11th Int'l Conf. Parallel and Distributed Computing (Euro-Par)*, volume 3648 of *Lect. Notes Comp. Sc.*, pages 1143–1152, Berlin, September 2005. Springer-Verlag.