COHESION - A microkernel based desktop grid platform for irregular task-parallel applications

Sven Schulz, Wolfgang Blochinger,
Markus Held and Clemens Dangelmayr

BibTeX

```
@article{Schulz2008,
      author = "Sven Schulz and Wolfgang Blochinger and Markus Held
                and Clemens Dangelmayr",
      title = "COHESION — A microkernel based Desktop Grid platform for irregular
               task-parallel applications",
      journal = "Future Generation Computer Systems",
      volume = "24",
      number = "5",
      pages = "354--370",
      year = "2008",
      issn = "0167-739X",
      doi = "https://doi.org/10.1016/j.future.2007.06.005",
      url = "http://www.sciencedirect.com/science/article/pii/S0167739X0700101X",
}
```

COHESION – A Microkernel Based Desktop Grid Platform for Irregular Task-Parallel Applications

Sven Schulz [1] Wolfgang Blochinger [*] Markus Held [2]
Clemens Dangelmayr

University of Tübingen, Symbolic Computation Group
Sand 14, D-72076 Tübingen, Germany

Abstract

We present Cohesion, a novel approach to Desktop Grid Computing. A major design goal of Cohesion is to enable advanced parallel programming models and application specific frameworks. We focus on methods for irregularly structured task-parallel problems, which require fully dynamic problem decomposition. Cohesion overcomes limitations of classical Desktop Grid platforms by employing Peer-To-Peer principles and a flexible system architecture based on a microkernel approach. Arbitrary modules can be dynamically loaded to replace default functionality, resulting in a platform that can easily adapt to application-specific requirements. We discuss two representative example applications and report on the results of performance experiments that especially consider the high volatility of resources prevailing in a Desktop Grid.

Key words: Desktop Grids, Grid Computing, Software Architecture, Peer–To–Peer Systems

[*] Corresponding author.
Email address: `blochinger@informatik.uni-tuebingen.de` (Wolfgang Blochinger).
[1] Supported by Deutsche Forschungsgemeinschaft (DFG) under grant BL 941/1-1
[2] Supported by a grant from the Ministry of Science, Research and the Arts of Baden-Württemberg (Az: 23-7532.24-4-18/1)

1 Introduction

Grid Computing is increasingly recognized as a major building block of the eScience vision. It provides researchers uncomplicated access to resources beyond the boundary of a single institution. Most importantly, Grid Computing can deliver unprecedented computing power.

In this paper we deal with Peer-to-Peer (P2P) Desktop Grid Computing. This discipline of Grid Computing integrates methods from high performance computing with state-of-the-art concepts from the realm of distributed systems. Desktop Grid systems harness underutilized resources of end-user computers for tackling computationally hard problems. This approach to Grid Computing is gaining momentum, since it is able to deliver considerable computing power at virtually no extra cost. Small scale installations, e.g. comprising the workstations of a department, and also large scale, Internet-wide approaches (also known as Global Computing) have been successfully implemented.

However, Desktop Grids differ significantly from traditional parallel systems. Particularly, resources exhibit a high degree of volatility and heterogeneity: Depending on the usage patterns of the participating desktop computers, resources with considerably differing capabilities join and leave the grid in an unpredictable manner. Delivering sustained computing power in such environments poses enormous challenges to system and application designers.

As a consequence, Desktop Grid applications are most often based on trivial parallelism. The problem at hand is decomposed into independent sub-problems, which can be farmed out for computation without further communication among the sub-problems. Extending the scope of the Desktop Grid approach towards more non-trivial parallel applications involves aspects on all levels of parallel system design. In this paper, we present our P2P Desktop Grid platform Cohesion (available from [1]). Our research aims at laying the system-level foundations for more tightly coupled parallel computations requiring complex interaction patterns among the participating nodes. The main target domain of Cohesion are irregular task-parallel applications. Typically, these applications employ fully dynamic problem decomposition based on a distributed task-pool execution model. In particular, we make the following contributions:

- We demonstrate how advanced P2P principles and techniques can be assembled to create a comprehensive collection of core functionality required for parallel computing.
- We show that a P2P approach to Desktop Grid computing also has a fundamental impact on the architecture of the system. We present an appropriate solution based on an industrial strength microkernel technology.

- We explain how sophisticated parallel programming models and application specific parallel frameworks can be realized on top of the abstractions and primitives provided by COHESION.

The remainder of our paper is organized as follows: In Section 2 we give an overview of related work. Section 3 discusses a P2P based execution model for irregular task-parallel applications. In Section 4 we identify requirements for P2P based Desktop Grid Computing. Section 5 gives a comprehensive description of the design, architecture, and implementation of COHESION. In Section 6 we present example applications and discuss the results of performance measurements.

2 Related Work

Grid computing has been a highly active area of research in the last decade. The textbook [2] provides a comprehensive overview of all major topics.

One main goal of Grid research has been enabling *virtual organizations* [3]. Within a virtual organization, organizationally-owned resources (possibly of considerable value) can be shared among autonomous and often geographically dispersed institutions without sacrificing local authority.

Desktop Grid computing [4] aims specifically at harnessing unused resources from non-dedicated end-user machines. This (largely orthogonal) discipline of Grid research has been motivated by today's pervasiveness of information technology and the resulting plethora of exploitable computing power [5]. According to the scale and the relationship among resource providers and resource consumers we can distinguish different approaches to Desktop Grid Computing. Computation exchange platforms (e.g. CompuP2P [9]) establish a symmetric relationship between resource providers and resource consumers by creating virtual markets for trading resources. A more asymmetric relationship between these parties is typical for volunteer (global) computing platforms (e.g. BOINC [10]) and for Enterprise Desktop Grids (e.g. Entropia [11]).

Both approaches to Grid Computing ultimately pursue the same goal, aggregating resources beyond local administrative domains. However, they face different requirements and constraints, e.g. target communities (limited trust vs. no trust) or nature of resources (high-end vs. end-user) [6].

System architectures for building virtual organizations (e.g. Globus toolkit [7]) must specifically deal with interoperability issues, like standardization of protocols and interfaces. It turned out, that these requirements can be effectively met by leveraging industry standards from the realm of web services

[8]. However, this approach results in rather complex systems and imposes high organizational requirements, e.g. highly qualified personnel. In contrast, architectures for constructing Desktop Grids must specifically reflect the high degree of resource volatility. Also, only little administrative overhead is acceptable, since typically no additional personnel is available for operating Desktop Grid installations. As a consequence, more lightweight and modular system architectures become mandatory, since they reduce software and runtime complexity and can also adapt to the prevailing dynamism.

Next, we classify several Desktop Grid platforms according to their basic architectural approach and subsequently, we elaborate on state-of-the-art techniques for building modular distributed software.

2.1 Client/Server Desktop Grid Platforms

Client/server or multi-tier Desktop Grid platforms employ a proven and well understood operational model. Thus, they have reached a considerable degree of maturity and stability and are particularly suited for commercial or mission critical applications. However, due to their centralized nature, programming models requiring complex interaction patterns among the participating nodes cannot be realized efficiently. Typically, such platforms exclusively support the *bag of tasks* or *master/worker* parallel programming model. As a consequence, client/server platforms are mainly suited for applications, which are based on trivial parallelism or for plain high throughput computing.

Well-known examples of this class of Desktop Grid platforms are BOINC and Entropia. Basically, these platforms are very similar. Differences can be observed concerning the client-side security model (signed code vs. native sandboxing), support of heterogeneous environments, and functionality for dynamically integrating applications.

In [12] a prototypical software stack for large scale distributed systems (LSDS) is presented, that specifically addresses resource volatility and respective application programming models. XtremWeb is a Desktop Grid platform which implements a subset of this architecture by a three-tier approach. It employs a coordinator for connecting clients and workers which is (currently) implemented in a centralized way.

2.2 P2P Desktop Grid Platforms

In recent years, several (experimental) platforms for high performance computing based on P2P principles have been described in the literature. Subsequently, we discuss some prominent representatives.

The JNGI [13] system is mainly targeted towards coarse-grained, embarrassingly parallel applications. It supports the master/worker programming model, thus being very similar to client/server platforms at the programming model level. However, the JNGI architecture focuses on extreme scalability and reliability. It pursues a self-organizing approach, taking advantage of the capabilities of the JXTA P2P platform. JNGI's architecture is based on several *peer groups* with distinct responsibilities, i.e. worker, task, and repository groups.

The primary goal of the Personal Power Plant (P3) Desktop Grid system [14] is to enable mutual and equal transfer of computing power between participating individuals. P3 provides master/worker as well as message passing based programming models. It is implemented on top of the JXTA P2P protocol suite.

While JNGI and P3 are built on unstructured P2P overlay technology, CompuP2P [9] is an example of a Desktop Grid platform, which relies on a structured overlay network based on distributed hash tables (Chord [15]). It creates dynamic markets of network accessible computing resources. For enabling resource trading, ideas from game theory and microeconomics are adopted.

Cohesion belongs to the class of P2P Desktop Grid platforms. Since it employs unstructured network overlay technologies, it is more closely related to systems like JNGI and P3. While other P2P based approaches focus mainly on achieving highest scalability, our research is primarily targeted towards realizing complex parallel programming models on Desktop Grids by leveraging P2P principles.

2.3 Modular Distributed Systems

H2O [16] is a distributed component framework that aims at removing the static binding between service deployer and resource provider. Although there are some similarities between H2O and Cohesion, there are also substantial differences. Both provide a framework for dynamically composing components or modules into applications. However, while H2O implements a proprietary approach, where components have to be assembled manually, Cohesion leverages a mature industrial-strength module system with advanced features like automatic component dependency resolution. A very fundamental difference is, that Cohesion is tailored to especially meet the requirements of P2P Desktop Grid applications. It simplifies application development by providing a comprehensive set of useful abstractions (e.g. groups, endpoints and tasks) and higher level distributed services. In contrast, H2O focuses on supporting componentization and flexible provider-centric module provisioning, while the implementation of higher-level services in H2O is completely up to the (application or third-party component) developer.

H2O components interact over RMIX [19], a programming model based on a generalization of Java RMI. It provides additional method invocation semantics (i.e. asynchronous and one-way invocations) and allows for transparently using different transport technologies. In contrast, Cohesion strives to support a multitude of programming models (with RMI(X) potentially being one of them) to support the diverse requirements of different applications. The concrete implementation of communication primitives (i.e. sending and receiving messages) is left to interchangeable substrates. Thus, every implemented programming model can be executed on every available substrate, creating a wide spectrum of possible design alternatives.

The Java version of the Harness Distributed Virtual Machine (DVM) [17] environment is based on H2O. It aims at providing a highly available, component-based, resource sharing platform for distributed metacomputing. High-availability in Harness is achieved through state sharing between kernels in virtual synchrony. However, this approach limits scalability and renders the DVM approach inappropriate for large P2P Desktop Grid systems. A refinement [18] of the Harness architecture allows for components to exist outside of the DVM context, resulting in better scalability. However, advanced features provided by the DVM, like failure recovery and event notification, are no longer available.

3 A P2P Based Execution Model for Irregular Task-Parallel Computations

Cohesion strives for pushing the limits of the Desktop Grid approach by enabling advanced parallel programming models. In this paper, we focus on task-parallel programming models targeted towards *Irregularly Structured Problems* (ISPs). ISPs are parallel applications, whose computation and interaction patterns are input-dependent, unstructured and evolving during the computation [20]. Prominent examples are discrete optimization and constraint satisfaction problems. In this section, we explain why the client/server model supported by many existing Desktop Grid Computing platforms is too restricted to efficiently parallelize ISPs and discuss how the arising problems can be tackled by introducing a P2P based execution model (cf. Figure 1).

Problem decomposition plays a central role in the design of parallel applications. It determines how the problem is to be divided into (sub-)tasks, which can be executed in parallel. Basically, problem decomposition can be carried out statically (i.e. tasks are identified and defined prior to program execution) or in a dynamic manner, where tasks are generated (on demand) at runtime.

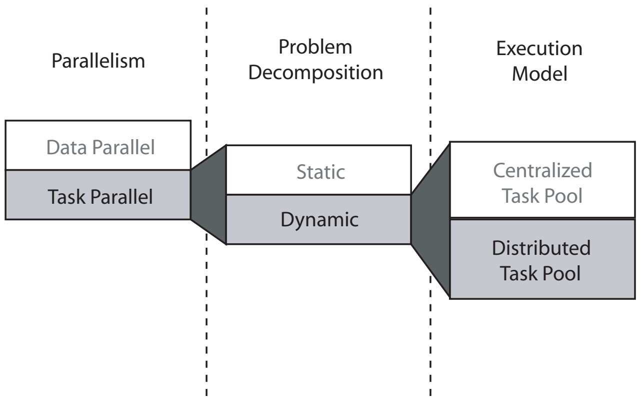


Fig. 1. Parallelization of task-parallel ISPs

In the latter case, tasks are explicit objects within the parallel program, which can be dynamically assigned to idle processors for execution. For ISPs a static approach to decomposition can result in significant processor idling, since a task's computational complexity typically can not be derived from program input. Thus, dynamic problem decomposition becomes mandatory.

Generally, dynamic problem decomposition requires explicit load balancing, i.e. tasks have to be assigned to processors at runtime. The *task pool model* decouples problem decomposition and load balancing by a data structure holding tasks that result from dynamic decomposition operations. It can either be organized in a centralized or in a distributed manner. In a centralized approach, a master node maintains a global task pool from which idle processors can fetch new tasks. To be able to serve task requests in a timely fashion, the master prompts active nodes to perform problem decomposition, whenever the size of the task pool falls below a given threshold. A drawback of this approach is, that the cost of maintaining an accurate view of all nodes' state becomes a sequential bottleneck for large numbers of nodes. Furthermore, tasks resulting from decomposition operations must first be transfered to the master node before they can be assigned to a worker [21]. Together, these shortcomings can seriously limit the overall efficiency.

In the distributed task pool model, a task pool is located on every node. Thus, problem decomposition and load balancing are accomplished autonomously by each node. Technically, the client/server model of a centralized task pool is replaced by a P2P model in which every node can be the source and sink of tasks. Load balancing can be implemented as sender (*work pushing*) or receiver initiated (*work stealing*) task transfers between local task queues (or combinations thereof). Because there is no global knowledge, distributed load balancing tends to be more scalable but less efficient than centralized approaches. However, since tasks resulting from local decomposition operations are transferred directly between nodes (and are not relayed by a master node), losses in efficiency are partially leveled out.

4 Peer-to-Peer Desktop Grid Computing

Although the term P2P is used differently in different contexts, there are several generic ideas and mechanisms such systems have in common. Typically, this includes a decentralized, self-organizing architecture, where autonomous and equal systems interact without any permanent central instance. In general, there are temporal but no fixed hierarchical client/server relationships between peers, that are tied to a specific purpose. Thus, P2P systems are often of ad-hoc nature and consequently exhibit an unpredictable and fluctuating overall performance. A comprehensive overview covering all facets of the topic is presented in [22]. For the purpose of our analysis, we confine ourselves to issues that we expect to have impact on parallel efficiency.

4.1 Challenges in P2P Desktop Grid Computing

Fine-grained Resource Control. Desktop Grid systems are running on non-dedicated hosts. They share resources (e.g. CPU cycles, network bandwidth, storage space) with other applications. In order to avoid interference with these applications while at the same time exploiting idle resources most effectively, a Desktop Grid system must allow for fine-grained resource usage monitoring and control. In Section 5.3, we describe how Cohesion integrates with the hosting environment to meet this requirement.

Limited Connectivity. In WAN scenarios, which are typical for Desktop Grid deployments, universal connectivity is hindered by widespread use of firewalls and network address translating (NAT) devices. Communication in systems with the client/server operational model is client-initiated and hence unaffected from this circumstance. However, P2P style communication requires mutual connectivity. To support this pattern, network segmentations induced by firewalls and NAT devices have to be bridged. Concerning NAT devices, there are several known traversal techniques [23], including relaying, connection reversal and UDP/TCP hole punching. For firewalls a common technique is to tunnel traffic over HTTP. These issues are actively addressed by emerging technologies like UPnP and IETF MIDCOM. However, these standards are still not widely supported. In Section 5.4, we show how we solved connectivity issues by leveraging existing P2P communication technology.

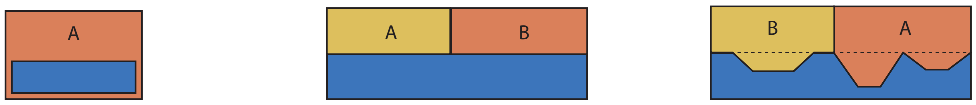
Ad Hocness. The spatial distribution of peers induced by network segmentations is accompanied by a temporal distribution. A host's availability is determined by the host system's usage pattern. Since these patterns are, in spite of some correlations (e.g. nodes located within the same timezone more likely share the same usage patterns), in general irregular with respect to different hosts, lifecycles of Desktop Grid nodes are largely decoupled.

Hence, a distributed computation in a Desktop Grid is much more dynamic than the same computation performed on a dedicated system. One fundamental factor determining the actual impact of dynamism on efficiency is, how fast node arrivals and departures are detected by the system. In Section 5.6, we demonstrate how COHESION supports ad hoc networks of nodes by providing a group abstraction with customizable QoS properties.

Heterogeneity. While dedicated parallel machines are typically comprised of identical or at least comparable subsystems, processors aggregated by Desktop Grids are far more diverse. Platform specifics can be hidden by utilizing a virtual machine. However, this measure can not deal with differences in host performance, which are typically very pronounced in Desktop Grids. Thus, differences in performance must be reflected in the systems execution model. As explained in Section 3, this issue is addressed by COHESION's support for fully dynamic problem decomposition.

Resource Volatility. *Availability* of a resource is the ability to perform its intended function over a period of time. It is usually expressed as the *availability ratio*, i.e. the quotient of uptime and total time. A reasonable refinement [24] of the term can be achieved by distinguishing between *host availability* and *CPU availability*. While the former is a binary value indicating, whether a host is not currently in a faulty state, the latter is usually defined as the fraction of CPU time consumed by the Desktop Grid application. The fluctuation in resource availability characterizes a distributed system's *volatility*. Accordingly, fault tolerance strategies will have to deal with two different kinds of volatility. *CPU volatility* originates in the deliberate shutdown of application instances by the COHESION application container. As described in Section 5.3, this may, for example, be actuated by user input or the orderly shutdown of the COHESION node itself. Applications may mitigate the impact of CPU volatility by exploiting the fact, that they are notified before an application's termination. *Host volatility*, on the other hand, arises from *host failures* (i.e. node and infrastructure failures) and usually does not allow for local compensation measures. However, their consequences can still be attenuated, as they are detectable by other nodes.

The actual influence of both kinds of volatility on performance and stability is heavily application dependent. Thus, the choice of a suitable fault tolerance strategy depends on application properties as well as on the underlying Desktop Grid's characteristics. In Section 5, we show how COHESION enables applications and higher level services to deal with both kinds of volatility by providing a managed application lifecycle and a distributed failure detection service.



(a) Monolithic (b) Middleware (c) Microkernel

Fig. 2. Evolution of the software architecture of Desktop Grid platforms

4.2 The Impact of P2P on System Architecture

Early Desktop Grid systems (e.g. SETI@home [25]) mixed up platform and application functionality in a non-detachable manner (cf. Figure 2(a)). Due to their monolithic and inflexible system design, these early systems were superseded by more flexible Desktop Grid platforms (e.g. BOINC) designed to support more than one application (cf. Figure 2(b)). The well understood client/server interaction patterns of applications deployed on these platforms still made it possible to provide a comprehensive set of platform functionality that was largely application independent.

The transition to P2P interaction patterns results in a multitude of options on every layer of the system. For example, the selection of an appropriate groupcast technique for a given application depends on a large number of factors, including network topology, expected communication load and required quality of service (QoS) properties. Generally, the design space for P2P Desktop Grid systems and applications becomes highly multidimensional and thus is considerably larger than the design space of conventional platforms. Consequently, providing a comprehensive toolbox serving all conceivable application requirements is no longer possible. Hence, a key purpose of next generation Desktop Grid platforms must be to provide a set of generic reusable components for common application aspects that may be replaced and supplemented by optimized extensions, which are contributed by applications.

However, this improved flexibility makes great demand on system design, since extensions may interact with core components of the system. Similar to the evolution in the field of operating systems, we think next generation P2P computing systems must be specifically designed to cope with application-specific customization at the system level. In Section 5.1, we describe how we attained this goal in Cohesion by explicitly supporting extension and customization of all major components through a design based on the *Microkernel* architectural pattern (cf. Figure 2(c)).

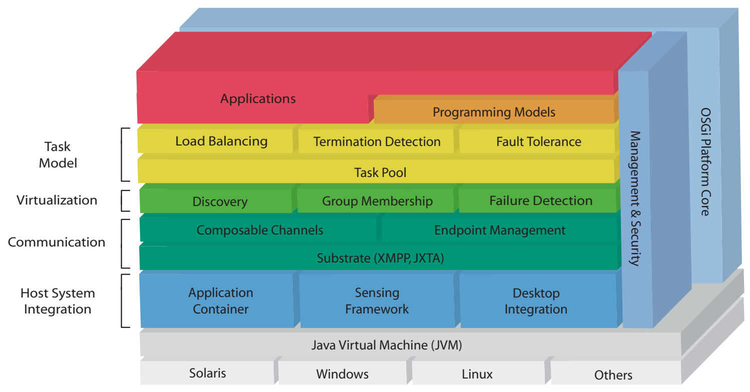


Fig. 3. COHESION's layered architecture

5 Design and Implementation of Cohesion

This section describes the design and implementation of COHESION and discusses how we addressed the challenges in achieving high efficiency for task-parallel ISPs. Figure 3 depicts the layered architecture of COHESION. Before we describe the functionality provided on each layer in detail, we give a brief top-down overview.

The **Task Model** layer provides a generic execution infrastructure on top of which various task-parallel programming models can be implemented. Tasks are put into a distributed task pool, which controls their lifecycle and enables task migrations between task queues. Lifecycle transitions are published as events, thus enabling the adaptation of the task pool's behavior to the requirements of particular programming models and applications. Thus, among other things, specialized load balancing and fault tolerance strategies suited for distinct Desktop Grid setups and applications can be easily plugged into COHESION.

Virtualization creates an abstraction layer between distributed resources and applications. This layer provides services to discover and organize resources, to monitor their availability and to coordinate their usage. Nodes become visible within the Grid by joining a *group*. A group's primary purpose is to define a scope for interaction. Resources contributed by group members are published and looked up using a discovery service. To maintain an accurate view of the available resources, an efficient failure detector is employed, that is used to detect unexpected node departures.

Communication in COHESION is highly configurable and enables higher-level virtualization services and programming models to instantiate custom communication primitives. The concept of *composable channels* allows to assemble communication channels from modularized quality of service (QoS) aspects, e.g. reliability, encryption or compression. Channels are established between *endpoints* that are first class objects, which can be published and looked up using the discovery service. The communication layer provides several unicast and groupcast endpoint implementations with different performance characteristics that may be complemented by custom ones.

Host System Integration provides functionality to embed COHESION into the hosting environment. A sensor-driven container-controlled application lifecycle ensures most effective resource utilization while at the same time preventing interference with the hosting environment's intended use. Sensors for CPU load, input devices (i.e. mouse and keyboard) and system time can be combined in arbitrary ways to control when applications can be executed.

Since COHESION is completely implemented in Java$^{\text{TM}}$and does not impose any other requirements on the hosting system, it is fully portable and runs on every platform for which a *Java Virtual Machine* (JVM) is available. A COHESION Grid is comprised of at least one edge node running the COHESION middleware and one or more (network specific) infrastructure nodes.

Due to their large scale and decentralized character, P2P Desktop Grid systems are considerably harder to manage than conventional distributed systems. COHESION specifically addresses this challenge by providing a comprehensive integrated management architecture, which is described in [26].

5.1 OSGi Platform Core

COHESION is explicitly designed to be modular and extensible with respect to almost any aspect of the system. This not only promotes COHESION's suitability as a research platform. As discussed in Section 4.2, we believe that this is the next natural step in the evolution of Desktop Grid middleware. With the ability to replace the generic implementations for all major abstractions, applications are free to deploy an implementation that best fits their needs.

A suitable architectural pattern for building such a reconfigurable platform is the *Microkernel* pattern [27]. It separates a minimum functional core from supplementary functionality and application-specific extensions and serves as a socket for plugging in and coordinating these extensions. Our implementation of the microkernel pattern is based on the *Open Services Gateway Interface* (OSGi) standard [28]. OSGi is a dynamic module system for the Java platform. It implements a service bus, that allows for dynamic discovery, assembly and online reconfiguration of modules and their interdependencies. By leveraging OSGi technology, we can profit from an industrial-strength framework for service-oriented component-based applications.

OSGi is designed as a layered framework. The module layer defines a classloading model, that substitutes the simple classpath model of the Java platform. This model allows for fine-grained control over the import and export interfaces of a module and forms the basis for OSGi's fully integrated security model with strict class space separation at module borders. The lifecycle layer introduces the notion of *bundles*. A bundle is a module with a lifecycle. Bundles may be dynamically installed, started, stopped, updated and uninstalled. OSGi provides declarative mechanisms to specify fine-grained dependencies between bundles at the Java package level.

Finally, the service layer provides a service mediation facility. Bundles may register service objects (i.e. plain Java objects decorated with descriptive properties) that may subsequently be discovered and bound by other bundles. When the registering bundle is stopped, all its registered services are automatically unregistered. To be notified of changes in service availability, bundles may register listeners. Based on a service description (usually an LDAP filter) the service bus selects matching service implementations and notifies the registered listeners. The concrete service implementation chosen by the bus service mediation protocol is completely transparent to the requesting bundle. OSGi's ability to keep implementation details private by restricting package visibility is used to prevent access to sensitive subsystems. This is of particular importance, since third-party bundles may contain malicious code.

Cohesion turns all of these features to profit: The bundle mechanism is used to subdivide the platform into manageable modules. Each component shown in Figure 3 is implemented as a bundle comprised of an API, i.e. the service interfaces, and at least one default implementation thereof. The ability to dynamically reconfigure the bundle set is leveraged to hot-deploy applications. Finally, the service bus enables Cohesion applications to replace out-of-the-box service implementations with application-specific ones.

5.2 Security

In P2P Desktop Grid Computing environments code is executed on geographically and organizationally dispersed peers. The lack of a central authority mandates that facilities are deployed to ensure that, on the one hand, no malicious code is executed on the peer and, on the other hand, the distributed computation's integrity is not tampered with. Cohesion is well-prepared to deal with common security threats by leveraging Java^{TM}and OSGi technology.

Cohesion implements a trusted code security model, where only certified applications can be deployed and executed within a *sandbox*. This precautionary measure is of particular importance for applications contributing system-level components, since these may compromise not only the application but the system as a whole. Sandboxing has been successfully employed in other Desktop Grid platforms [29] to prevent abuse of participating hosts and alteration or disclosure of application data and code. COHESION adopts this approach by using the sandbox provided by the JVM combined with the OSGi security infrastructure. The latter provides a role-based security facility that complements the code- and origin-based permission model of the Java platform. Furthermore, OSGi allows for dynamic modification of the security policy. Thus, we can prompt the user, when a suspicious action is attempted by an application.

Security has been taken into account on other layers of the COHESION protocol stack. Communication can be secured by using Java cryptographic streams, that are wrapped around raw transport channels using COHESION's *composable channel* service. On the virtualization layer, COHESION groups can be secured, so that only nodes with suitable credentials are able to join. Since resources published in a group can be discovered and instantiated by group members only, they are effectively shielded from unauthorized access. To implement this feature, COHESION relies on the facilities provided by substrate technologies.

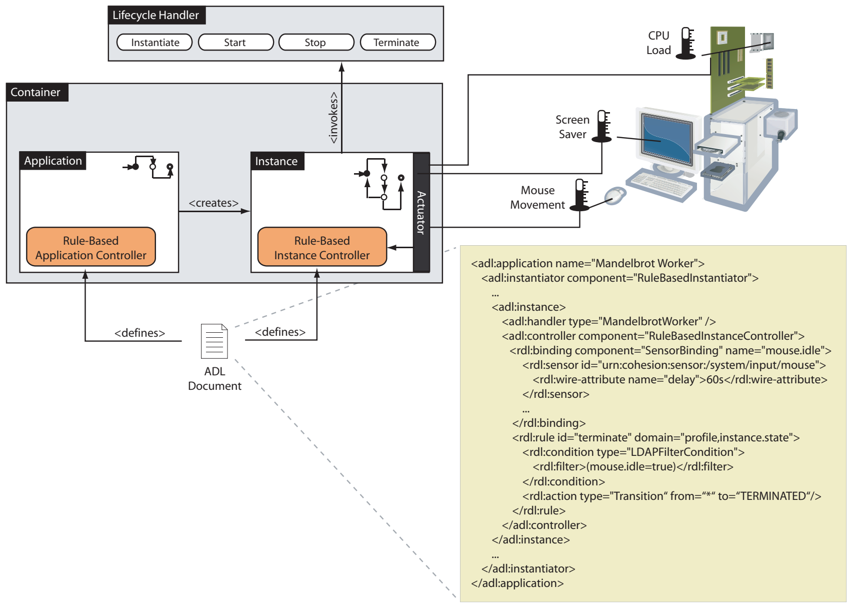
5.3 Host System Integration

COHESION is embedded into non-dedicated host systems sharing resources like CPU cycles, network bandwidth and disk space with other user processes. On the one hand, COHESION should execute in a minimal invasive way, that does not interfere with these user processes, thus ensuring host system sovereignty. On the other hand, idle resources should be utilized most effectively to optimize application efficiency.

In contrast to other platforms, COHESION applications are not executed permanently as a background process or only when the system screensaver is active. The first has proven to be well-suited for primarily CPU-bound problems. However, for applications with high memory consumption or I/O subsystem load, the user experience may be compromised by penalties caused by resource exhaustion, like excessive paging/swapping (thrashing). While perfectly shielding the user from impairment, the latter is too inaccurate, especially with the advent of multi-core processors, resulting in suboptimal resource utilization.

Instead, COHESION dynamically reacts to changes in host system resource availability. For that purpose selected host system parameters (e.g. mouse/keyboard activity or CPU load) are permanently monitored using a pluggable sensing framework. When user-defined conditions are satisfied, applications are dynamically activated or deactivated by the system.

Since applications in COHESION can contribute extensions and drop-in re-



```xml
<adl:application name="Mandelbrot Worker">
  <adl:instantiator component="RuleBasedInstantiator">
    ...
    <adl:instance>
      <adl:handler type="MandelbrotWorker" />
      <adl:controller component="RuleBasedInstanceController">
        <rdl:binding component="SensorBinding" name="mouse.idle">
          <rdl:sensor id="urn:cohesion:sensor:/system/input/mouse">
            <rdl:wire-attribute name="delay">60s</rdl:wire-attribute>
          </rdl:sensor>
          ...
        </rdl:binding>
        <rdl:rule id="terminate" domain="profile,instance.state">
          <rdl:condition type="LDAPFilterCondition">
            <rdl:filter>(mouse.idle=true)</rdl:filter>
          </rdl:condition>
          <rdl:action type="Transition“ from=“*“ to=“TERMINATED“/>
        </rdl:rule>
      </adl:controller>
    </adl:instance>
    ...
  </adl:instantiator>
</adl:application>
```

Fig. 4. COHESION's sensor-driven container-controlled application model. An ADL document is used to define under what conditions an application is executed.

placements for generic platform components, it is crucial to protect the system from uncontrolled application behavior, that may render the system state inconsistent. Thus, COHESION mandates a strict lifecycle for applications contributing such components. This lifecycle, modeled as a finite state machine, is enforced by an application container. While the set of available states is fixed (i.e. FETAL, INITIALIZED, RUNNING, SUSPENDED and TERMINATED), the transition conditions are customizable through pluggable controllers. Prior and after each transition, an application-specific callback handler is invoked. Thus, application logic can react to state changes of the underlying application instance. Controllers are employed on two different logical levels: while *application controllers* are responsible for deploying, undeploying and instantiating applications, *instance controllers* are used to start, suspend, resume and terminate application instances. Thus, every aspect of an application's lifecycle can be controlled by deploying and configuring a suitable pair of controllers.

The most advanced controller family of COHESION is based on a rule engine. By incorporating host environment embedded sensors, transition rules, which are declared in our *Application Description Language* (ADL), can be used to define application control logic, that respects host system sovereignty, while at the same time exploiting idle resources most accurately (see Figure 4).

5.4 Substrates

COHESION is designed to be network agnostic. It can be used on top of any network technology that satisfies a minimal set of constraints. To abstract from the specifics of the underlying network, we introduce the notion of a *substrate*. It consists of

- a **group implementation** with an (at least) weak partial group model, that is used to instantiate the root group,
- a **unicast endpoint** that provides at least connectionless channels with unreliable message delivery, from which channels with higher-level quality of service characteristics can be derived, and
- optionally, a **groupcast endpoint** that takes full advantage of the features of the underlying network technology and thus can be used as an efficient drop-in replacement for the generic groupcast implementation provided by COHESION that simply unicasts messages to all group members.

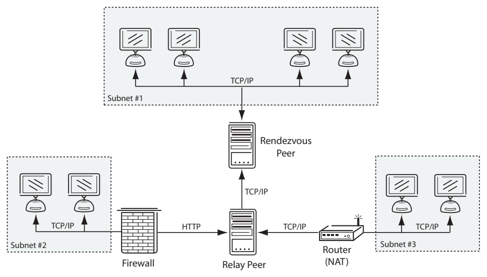
Substrates may be environment specific or serve a special purpose. For example, an approach based on a star topology can be used to route all messages through a central gateway, thus providing insight into the message flow for debugging or performance evaluation reasons. Up to now, we have implemented two substrates: a scalable one, that is based on JXTA and a more efficient one based on XMPP.

While JXTA is considered to be a good choice for implementing distributed computing platforms [30], performance studies reveal weaknesses in the current Java reference implementation concerning pipe latency and throughput for small messages [31], reliability of TCP connections [32] and rendezvous network stability [33].

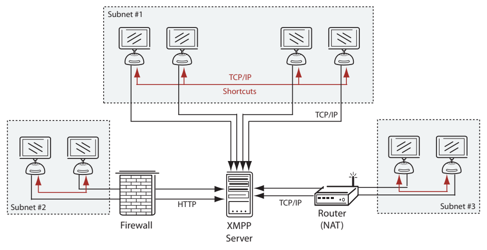
XMPP implements a more centralized scheme, hence limiting scalability, but with superior performance. Nevertheless, XMPP servers have shown to scale up to tens of thousands of concurrent clients (at least for moderate message rates) [34]. Our XMPP substrate uses TCP/IP connections to bypass the XMPP server for link-local communication (i.e. within the same segment), thus permitting higher overall traffic.

5.4.1 JXTA

JXTA [35] is an open source project initiated and maintained by Sun Microsystems. It is based on an open XML protocol stack, which allows any two devices on a network to exchange messages despite of the underlying network topology. This is accomplished by establishing a virtual overlay network



(a) JXTA overlay network with rendezvous and relay infrastructure peers.



(b) XMPP network with TCP/IP shortcuts between nodes located in the same subnet.

Fig. 5. Cohesion network substrates

that allows a peer to transparently interact with other peers even across network boundaries induced by firewalls, NATs or non-interoperable transports. JXTA is the most mature P2P communication framework currently available that allows for true decentralized communication.

The JXTA virtual overlay network (see Figure 5(a)) employs two kinds of special purpose peers. *Rendezvous peers* have an optimized routing mechanism that allows an efficient propagation of messages received from edge peers. Therefore, a loosely consistent network of rendezvous peers is maintained. Peers that are behind firewalls or NAT systems still can take part in the JXTA network by using the services of a *relay peer*. Typically, this is done over protocols like HTTP that can traverse introspecting firewalls.

JXTA groups are, similarly to those of COHESION, used to scope communication and to establish secure partitions within the unstructured set of participating nodes. However, JXTA does not provide an explicit facility to get a nodes view on the groups membership. Instead a simple discovery service can be used to query for peers within a peer group. Since this approach produces fluctuating membership views due to flaws in the reference implementation of the discovery service, we use our substrate agnostic generic group model that is described in Section 5.6.1. While unicast communication is realized on top of JXTA sockets, groupcast communication is based on JXTA propagate pipes that use IP multicast, thus offering support for efficient broadcasts on local subnets.

5.4.2 XMPP

The *Extensible Messaging and Presence Protocol* (XMPP) is an open, XML-based protocol for real-time instant messaging. As the core protocol of the *Jabber Instant Messaging and Presence* technology, XMPP is deployed on thousands of servers and is used by millions of people worldwide.

XMPP is a client/server architecture (see Figure 5(b)). Hence, even for point-to-point communication the XMPP server is required to route messages to their destination node. Consequently, the server turns out to be a bottleneck for large numbers of communicating clients. To overcome this limitation, we have implemented an extension that allows for establishing direct TCP/IP connections (*shortcuts*) between nodes within the same segment. Since messages delivered over shortcuts bypass the XMPP server, we can achieve considerably higher message rates. TCP endpoints used to establish shortcuts are advertised and discovered by embedding them into *vCards*, which are defined by the *Internet Mail Consortium* (IMC). The XMPP unicast endpoint implementation periodically checks whether the recipient of a message is reachable via the shortcut, falling back to ordinary server-based delivery if not.

Cohesion groups are mapped one-to-one to XMPP *Multi-User Chat* [36] rooms (MUC). A MUC room is populated by a number of members, its *occupants*. The list of occupants is maintained on the XMPP server hosting the MUC room. Any update is immediately pushed to all members of the room. Thus, XMPP-based Cohesion groups are both accurate and efficient even in the absence of an explicit failure detection component. XMPP groupcasts are implemented by facilitating MUC's ability to broadcast messages to all room members. Both, the XMPP-based group model and the groupcast implementation, are drop-in replacements for the generic substrate-agnostic implementations described below.

5.5 Communication

Cohesion nodes communicate by exchanging SOAP (*Simple Object Access Protocol*) messages over *channels*. SOAP is used as a lightweight, structured and extensible message format. Utilizing an XML message format not only boosts interoperability, but also ensures operability in the presence of introspecting firewalls.

Cohesion's *composable channels* are an instance of the *Decorator* [37] design pattern: Channels of arbitrary complexity can be constructed from comparatively simple reusable components. Each component encapsulates a certain aspect of communication (e.g. reliability through a sliding window protocol algorithm, encryption or compression). This modularization of communication aspects fosters reuse and reduces complexity. The composition process is implemented as a service. *Compositors* may be registered by provider bundles. Following the *Chain of Responsibility* [37] pattern, each registered compositor contributes to the composition by wrapping a proxy channel around the raw channel that satisfies a subset of the yet unresolved set of constraints. The resulting channel and the remaining unsatisfied constraints are passed on to

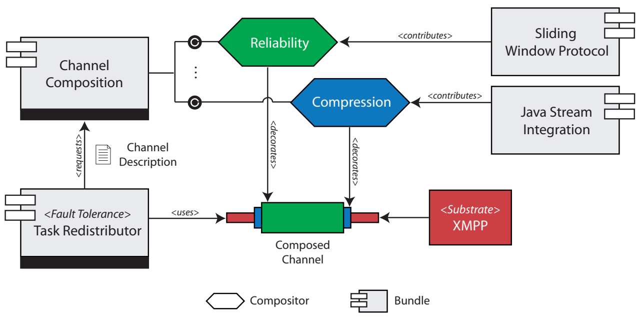


Fig. 6. Channel composition from encapsulated communication aspects contributed by bundles.

the next compositor. If any constraints are still unsatisfied after the last compositor has been engaged, the application is notified that the request can not be satisfied. Figure 6 illustrates the composition process for a reliable channel with transparent compression. By using channel composition, implementors are free to assemble channels from a repository of reusable components with exactly those QoS properties they actually need, while at the same time minimizing system load by avoiding unnecessary protocol overhead.

Channels are established between *endpoints*. COHESION provides a range of unicast and groupcast endpoints with different semantics. They are contributed as extensions to and managed by an endpoint management facility. There are two conceptually different types of endpoints in COHESION: While ordinary endpoints are resources, and are published and discovered using the discovery service, *a priori* endpoints, like groupcast endpoints, are available immediately after the hosting group is joined. The reason why the former must be discovered is, that they usually depend on information which is available to the publisher only (e.g. IP address and TCP port or a JXTA pipe identifier).

To support a wide spectrum of different requirements, we provide several groupcast implementations featuring different transmission characteristics: a generic one that uses unicast endpoints to deliver messages, an epidemic one that is bandwidth conservative as it injects payload into external messages to reduce overhead introduced by non-payload message parts, and an implementation of *Bimodal Multicast* [38], a randomized protocol with probabilistic reliability properties. While the aforementioned groupcasts are available irrespective of which substrate is used, our substrates deploy optimized implementations that take advantage of the underlying communication infrastructure (see Section 5.4).

By specifying abstract constraints, client bundles can – without any knowledge about the actual implementation – select an endpoint implementation that best fits their needs. This is crucial, since an inappropriate choice may have a considerable impact on the scalability and performance of higher level functionality.

5.6 Virtualization

Through virtualization, higher level services (e.g. workload balancing) can deal with resources as first class objects. In COHESION parlance, virtualized resources are called *entities*. Examples of entities are groups and endpoints. Entities are announced to the system by publishing an XML description within a distributed directory. Once discovered, a local proxy object is derived from the description, which can be used to interact with the remote entity. In contrast to the discovery service of JXTA, COHESION leverages *XQuery* [39] to allow nodes to lookup resources with maximized expressiveness, thus reducing bandwidth consumption by minimizing unnecessary transmission of entity descriptions.

The efficiency of resource utilization, and hence of the parallel computation itself, heavily depends on an up-to-date view of the available compute resources. Since resources are contributed by nodes, this translates to the necessity to quickly and accurately detect node arrivals and departures. Detection speed, i.e. how long it takes until a healthy node becomes visible or a faulty node is detected, and accuracy, i.e. how much the reported differs from the actual membership, are aspects of the *efficiency* [40] of the group model and failure detection scheme employed. We could have incorporated a monolithic approach (i.e. based on a lease mechanism), combining group membership and failure detection. However, this would have introduced a direct coupling between detection speed and accuracy, limiting the overall efficiency. Hence, group membership and failure detection are separate concepts in COHESION. Since group membership is maintained on the server and propagated instantly to group members in XMPP, our XMPP-based substrate deploys a custom solution that is very efficient without using a failure detector at all.

5.6.1 Group membership

In COHESION, *nodes* become visible as entities by joining a *group*. Groups are entities too and thus may be published and discovered using the discovery service. A group's conceptual purpose is to establish a logical partition within the node set and to define a scope of interaction (e.g. for communication through groupcast endpoints). Since groups are hierarchical, programmers may express relations among groups (parent/child, ancestor/descendant, sibling, etc.) in accordance with the application's domain model. This feature is frequently used throughout the components of COHESION. For example, a separate group is created at the task model layer for each calculation, thus promoting isolation (in terms of security and fault tolerance).

A group's properties are determined by the implemented group model. In

Cohesion, such models may be contributed as extensions and are bound to concrete groups at runtime based on constraints provided on creation. Thus, higher level services are able to select a group model, whose properties best fit their needs. Cohesion provides two group implementations with different QoS characteristics out-of-the-box.

With Scamp [41], Cohesion supports a fully decentralized, self-organizing membership protocol that establishes a partial view of size $O(log(n))$ on each member node. The resulting high scalability is required for supporting large numbers of members in the Cohesion root group (which consists of all available nodes).

For efficiently supporting fine-granular parallel computations, we also need a group model with complete local views. This is an exemplary use case for Cohesion's extensibility and customizability. We can deploy a simple view-complete group membership algorithm based on subscriptions within calculation groups. When a node joins a group, it groupcasts a subscription message that uniquely identifies itself. On receipt, remote nodes add the respective node to their membership list. To populate the membership view of the subscribing node, $O(log(n))$ randomly chosen nodes respond with a copy of their membership list. More precisely, each node emits such a seed, with probability $O(log(n))/n$. Since the set of seeding nodes may be empty and the employed groupcasts implementation is unreliable for efficiency reasons, nodes may miss initial subscriptions. Thus, subscriptions are reemitted periodically. Without volatility and supposing no node is permanently isolated from the others, all nodes eventually share the same complete view on the group's membership. To unsubscribe, departing nodes groupcast a sign off message. On receipt, remote nodes remove the issuing node from their membership list. Again the sign off message may get lost. Thus, subscriptions are valid for a given finite period only. If a subscription expires, the node is removed from the membership list. To tolerate stochastic message losses, subscriptions are emitted more than once during the validity period of a subscription. If global membership remains stable, local membership views converge and become eventually complete. With n member nodes, we have $O(n)$ groupcasts per time unit resulting in $O(n^2)$ message complexity on the link layer, for any substrate that provides a groupcast with $O(n)$ message complexity.

5.6.2 Failure detection

Failure rates in desktop grids are considerably higher than in common parallel environments. To limit the impact on resource utilization, speed as well as accuracy of failure detection becomes key.

In Cohesion, host availability is defined by a node's membership in a group.

However, if a node crashes or becomes isolated by faulty network links, the node's membership would not be canceled through explicit unsubscription by the group membership protocol. Thus, the delay between node departure and the detection thereof is determined by the validity period of subscriptions. However, decreasing the subscription validity period by a factor of f means an increase of $O(fn^2)$ in message complexity. Hence, having a complementary failure detection protocol with lower message complexity is beneficial. Thus, we use a dedicated *failure detection component* to detect ungraceful node departures.

Our default failure detection protocol has been adopted from the SWIM system [42]. Nodes are monitored through an efficient P2P periodic randomized probing protocol. Neither detection speed nor message load per member vary with group size. False positives (i.e. nodes that are wrongly considered faulty) are reduced by first suspecting a participant node before declaring it as failed. The detection delay is bounded by $\Omega(RTT_{wc})$, where RTT_{wc} is the worst case round trip time (considered non-faulty). The physical layer message complexity is $O(1)$ per healthy and $O(n)$ per faulty node and round, resulting in an aggregated asymptotical worst-case complexity of $O(n^2)$. Although the asymptotical complexities are identical, the number of messages sent by the detector is considerably smaller, since faults are — albeit frequent — still the exceptional case.

5.7 Task Model

Cohesion adheres to a clear conceptual distinction between task model and programming models. This simplifies the adaptation of advanced programming models, for their implementors will not have to deal with lower level details of task enactment and migration. Consequently, from the Cohesion task model's point of view, programming models are applications themselves.

A task model, which is suitable for programming models that support dynamic decomposition, has to deal with several nontrivial issues. In particular, spawn relationships and dynamically created data dependencies between tasks have to be modeled, since tasks may be created anywhere and anytime. Our task model defines a task lifecycle managed by a distributed task pool. Once a task is added to the (local) pool, it will eventually be executed in the Desktop Grid. In addition to the distributed task pool, load balancing and fault tolerance are capsuled within distinct bundles.

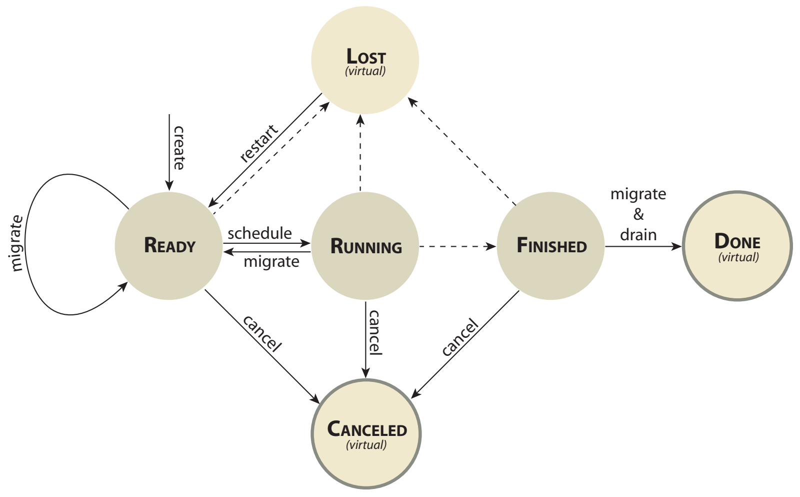


Fig. 7. A Cohesion task's lifecycle.

5.7.1 Task Pool

Tasks are managed by a distributed task pool formed by the local task pools running on each Cohesion node. A local task pool includes a dequeue of ready tasks and a configurable set of worker threads, which execute the tasks. A Cohesion group is used to define a scope for interactions between local task pools (e.g. a group's view is used to select partners for load balancing actions) and to isolate pools from each other to support concurrently executing computations.

Logically, a local task pool manages the tasks' lifecycle and notifies subscribers of lifecycle transitions in an *Observer* pattern style. A task's lifecycle (see Figure 7) consists of its creation, possibly several migrations, its completion and finally the delivery of its result. Tasks are marked done when their result is delivered, can be canceled before completion, and may be temporarily lost due to host failures or incomplete migrations. For tasks being in such a *virtual* state, there is no actual task instance, although they are still represented within the system by means of a unique task identifier to support fault tolerance mechanisms or to avoid execution of descendants of already canceled tasks.

Every task is associated with two distinguished nodes: its *source node*, where it has been created, and its *drain node*, which will accept its results. While ready tasks may be migrated for load balancing purposes, finished tasks will migrate to their drain node. Due to limitations of serialization in Java, Cohesion supports *weak migration* only. In contrast to *strong migration*, where the whole memory image (including the current state of the stack, the value of the program counter and all reachable objects) is sent to the destination site, weak migration transfers the serializable state of an object (i.e. all referenced objects) only and requires that the migrating object has agreed to and actively supports its own migration.

As discussed in Section 4, applications are shut down when the CPU is no longer available. In this case, the local task pool migrates remaining tasks to randomly chosen remote pools, thus reducing the impact of CPU volatility. In contrast, host volatility is more difficult to handle since host failures are unpredictable and happen instantaneously. Thus, affected nodes are not able to react as in the case of graceful application termination. They even cannot unsubscribe from joined groups and thus remain a part of the membership lists of remote nodes. However, since the remaining nodes will eventually be notified of the host failure by the COHESION failure detector, lost tasks, that were located at the faulty host, can be revived by a fault tolerance module. For that purpose, tasks may define a special restart behavior (see Section 5.7.3).

5.7.2 Load Balancing

Conforming to COHESION's design principles, load balancing is capsuled in an interchangeable module. We provide two default implementations. Depending on infrastructure and application characteristics custom approaches can be employed.

In smaller, homogeneous networks a randomized work stealing strategy can be used, where idle nodes periodically send queries to randomly selected peers. Non–idle nodes answer by transferring some of their ready tasks. For other cases load balancing strategies can use COHESION groups to model the underlying Grid topology. One such approach is the combination of a random stealing strategy within local COHESION groups with a random pushing strategy within a global COHESION group. A node with a non–empty ready dequeue, having not received a stealing message for some time, may ask nodes outside of its local group, whether they are idle and transfer some of its ready tasks on the receipt of a positive answer. This strategy makes use of COHESION's ability to combine group views via set operations. It can be beneficially applied in the case of segmented networks with inhomogeneous latencies and bandwidths.

5.7.3 Fault Tolerance

COHESION provides a default fault tolerance module implementing an extensible task restarting strategy to deal with lost tasks. On the creation of a task, its source node sends a copy to the drain node, which records the task as well as its location. Once a task's drain node detects a fault at the task's current location (which may have changed since its creation due to one or more migrations), it will invoke the `restart` method on the retained copy of the lost task and add it to the local task pool. Since a node on a faulty host is no longer able to process results, all tasks depending on it as their drain node are canceled.

To support application specific extensions of the basic scheme, Cohesion employs the *Visitor* pattern. If a restart operation is initiated, each registered visitor may modify the list of tasks about to be restarted as well as the tasks themselves. As described in Section 6.2, this feature can be used to implement an application-specific checkpointing scheme.

6 Applications

In this section, we evaluate our approach in the light of two applications exhibiting a high degree of irregularity — *Mandelbrot Sets* and *Discrete Optimization*. Moreover, we demonstrate how programming models suitable for ISPs can be implemented on top of Cohesion's task model.

6.1 Experimental Setup

As explained in Section 4, there are many factors influencing the performance of a Desktop Grid system. Conducting live tests is not a promising approach to provide insight into the effects of volatility, since measurements are not reproducible. Using network simulators is not an option either, since application behavior itself is a key aspect of the system's behavior. Thus, our testing strategy is double-tracked. On the one hand, we conducted a series of measurements under artificially generated volatility within a controlled and homogeneous environment. For this setting we employed the Mandelbrot Set application, because it exhibits a high degree of irregularity and at the same time does not suffer from work-anomalies. (In this context, the term work-anomalies means that the total amount of work differs significantly between the sequential and the parallel computation and/or between several parallel computations of a problem instance. For example, this behavior can often be observed for heuristic graph search methods [43].) Thus, Mandelbrot Sets are well-suited to provide meaningful insights into the performance characteristics of Cohesion under resource volatility.

On the other hand, a test series with the *Traveling Salesperson* problem (TSP) and a larger number of heterogeneous nodes gives evidence to the efficiency and scalability of Cohesion under real-world conditions. Our testbed consisted of hosts from two pools (see Table 1). While hosts within the same pool were interconnected by a 100 Mbps Fast Ethernet network, communication between hosts from different pools is carried out over a campus WAN.

Type	**Hardware**	**OS**
I	3 GHz Pentium 4 (1 GB RAM)	Windows XP
II	3.06 GHz Celeron (1 GB RAM)	SUSE Linux

Table 1
Host types used for experiments

For evaluating COHESION's behavior in a volatile environment, we had to simulate the effects of CPU and host volatility. To control CPU availability, we leveraged COHESION's application lifecycle support. Nodes were configured to run the respective application for a given time-period only. When this time is elapsed they query a coordinator node, whether to shutdown orderly or abruptly. The former results in controlled termination of the application and actuates the migration of a node's tasks to randomly chosen nodes. The latter simulates a host failure. Thus, no measures are taken to compensate loss of uncommitted work. After termination a node is immediately restarted. Since the time penalty for restarting COHESION is small, we can retain the notion of parallel efficiency, thus focusing the analysis on the effects of volatility itself. Even in the case of an orderly shutdown, work may partially be lost. Thus, the average efficiency will certainly drop below its value in the non-volatile case. A host failure, however, prevents any measures by the failing node. Thus, the impact of host volatility on efficiency will be more severe. Using a coordinated approach for controlling volatility allows us to measure the effects of arbitrary mixes of shutdown and host failure events.

6.2 Multithreading Programming Model

Based on the task model presented in Section 5.7, COHESION provides an object–oriented implementation of the strict multithreading parallel programming model [44]. Strict multithreading is a powerful tool for the development of distributed programs based on dynamic problem decomposition.

In multithreading programming models, threads can create (or *spawn*) new threads. A multithreaded computation is called *fully strict*, if a thread may only send results to its parent thread. Thus, fully strict multithreading is very similar to programming with asynchronous procedure calls. The class of *strict multithreaded* computations is a superset of fully strict computations. In strict multithreading a thread's addressee may either be its parent or its parent's addressee and threads can deliver several results. This can be modeled by the concept of *thread groups* [45]. A thread group is a set of threads delivering their results to the same data sink. A thread can own any number of thread groups and query their results.

Cohesion supports strict and fully strict multithreading. Threads can be explicitly added to a thread group (*fork* primitive), or implicitly added to its parent's thread group (*hyperfork* primitive). They are implemented as Cohesion tasks, while thread groups are modeled as objects, which are owned by a thread. Results are delivered to a queue held by the thread group. A thread group's owner can query its results via the *join* primitive.

Multithreading task identifiers consist of two vectors. One vector represents the task's position within the spawn tree. The other vector counts the number of restarts due to fault tolerance measures. Thus, a thread is uniquely identified. Threads may query properties of the local task pool (e.g. the ready dequeue's current size) to decide upon the necessity of a decomposition step.

6.2.1 Mandelbrot Sets with Strict Multithreading

A *Mandelbrot set* is a fractal subset of $\mathbb{C}$. A number $(x, y) \in \mathbb{C}$ belongs to a Mandelbrot set if it is *quasi–stable*, i.e. the absolute value of a specific function does not exceed some limit after any number of iterations. In visualizations of Mandelbrot sets a pixel's color value is determined by the number of iterations for which this condition holds. Mandelbrot sets are a typical example of irregular problems, because the iteration depth of pixels and thus the computation time of pixel rows may vary heavily and cannot be predicted [46].

Our parallel approach advances row–wise. Initially, a certain amount of tasks (comprising several rows) are generated using the fork primitive. When nodes run idle, a running task dynamically creates a new task (employing the hyperfork primitive), which gets half of the rows not yet computed. Each completed row is sent to the initial task (the thread group's owner). Hence, the initial node can keep track of a computation's progress. This enables the implementation of an implicit checkpointing mechanism by registering an appropriate visitor with the task restarter module (see Section 5.7.3). This visitor will adapt the restarted tasks' row sets according to the currently open rows. Returning results row–wise also helps evening out the network load on the thread group's node. In the case of an orderly shutdown, remaining tasks are transfered to randomly chosen peers. A pointer to the current row is kept in a field of the running Mandelbrot thread, thus enabling its migration, though the row currently being computed is lost.

If the frequency of incoming results drops below a threshold, new threads are created from the list of open rows (eager scheduling). This ensures the application's progress under high failure rates. Additionally, the initial node groupcasts a message with the list of open rows and an identifier of the current computation within constant intervals. This helps preventing excess computation resulting from fault tolerance measures and outdated tasks. The initial node detects termination of the parallel computation, when the list of open sub-problems becomes empty.

6.2.2 Performance Evaluation

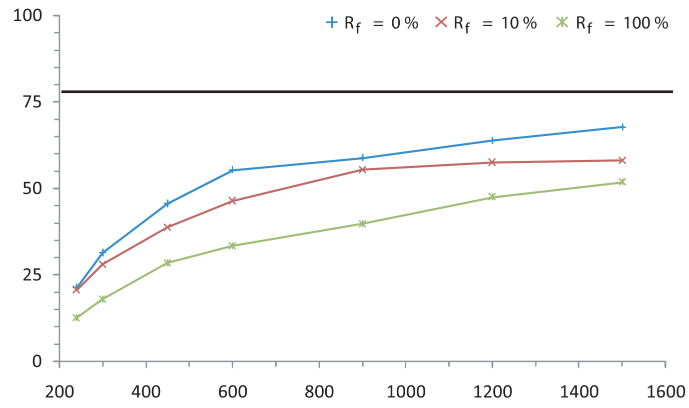
This section presents an analysis of the performance results obtained for our Mandelbrot application in a controlled yet volatile Desktop Grid under worst-case conditions. Our testbed consisted of 48 COHESION instances running on Type-I nodes. While the sequential runtime of the problem instance used throughout the test series was 49506 seconds (approx. 13 3/4 hours), the parallel runtime in a non-volatile setup was 1324 seconds (approx. 22 minutes) resulting in a parallel efficiency of 78% (or a speedup of 37.4).

Figure 8(a) shows the parallel efficiency achieved for increasing resource volatility and different ratios (R_f) of host and CPU volatility (i.e. $R_f = 10\%$ means one out of ten events is due to host volatility). We conducted 10 test runs for each of a total of 21 different configurations.

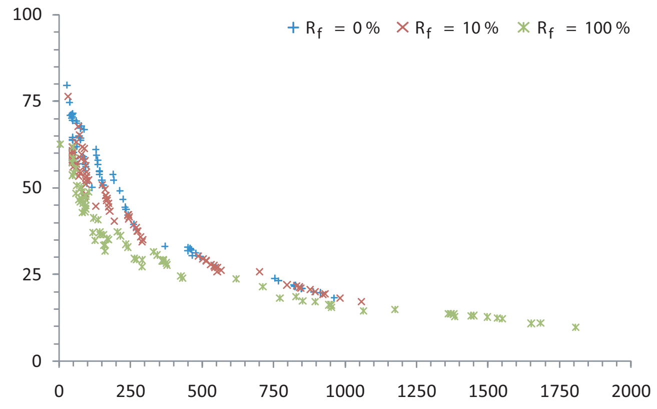
Every loss of a task results in the loss of a portion of work already done. Thus, it is not surprising that the efficiency is more and more degrading when the event rate per node increases. The significant differences in efficiency (between 30% for $T = 1500s$ and 74% for $T = 300s$) obtained for varying event type ratios stem from the fact, that the amount of lost work is different when nodes are orderly shutdown and nodes are terminated by host failures. While in the case of CPU volatility, only the row currently being processed is lost, the impact of host failures is more severe, since the task as a whole is restarted on another node. This result indicates that application-specific compensation measures on application termination are quite effective.

The fact that in the case of pure CPU volatility losses in efficiency are that pronounced can be attributed to penalties resulting from the time necessary for redistributing tasks from the local task queue prior to application termination and startup effects, i.e. the idle time before a first task is successfully stolen after a restart.

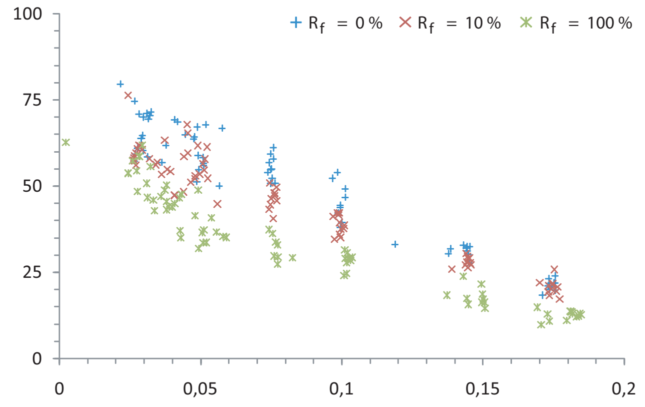
Figure 8(b) and Figure 8(c) clearly show that the overall number of events and the frequency with which they occur heavily influence the overall efficiency. Furthermore, they corroborate the observation that the impact of host volatility is more severe. Note that the worst-case character of our tests is evident from Figure 8(c): A maximum of 0.18 events per second is far beyond what is to be expected in a similar sized live Desktop Grid.



(a) Parallel efficiency (%) against per node event intervals (s)



(b) Parallel efficiency (%) against overall number of events



(c) Parallel efficiency (%) against event frequency (1/s)

Fig. 8. Efficiency of *Mandelbrot* in a volatile environment.

6.3 Aspect-oriented Discrete Optimization

In this section, we show how aspect-oriented techniques help to execute existing code for discrete optimization in a Cohesion Grid without any modification of the sequential code base [47].

6.3.1 Parallel Discrete Optimization

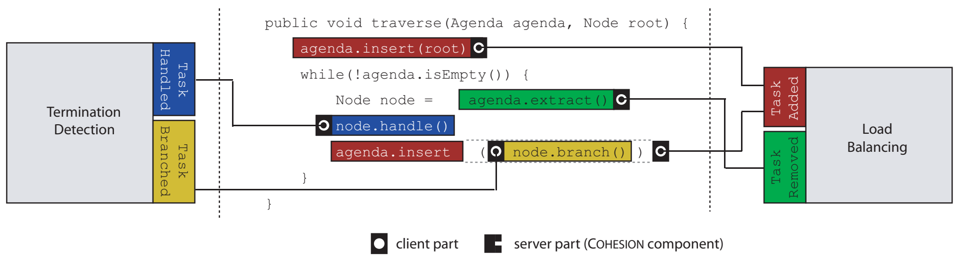
Discrete optimization is concerned with finding an element x_{opt} out of a finite discrete set X defined by certain constraints. In most cases, this means minimizing a function f with $f(x_{opt}) = min\{f(x)|x \in X\}$. Using *state space graphs*, elements of X can be represented as paths in a graph. An example is the *Traveling Salesperson Problem* (TSP), where paths represent partially constructed tours, starting from the empty tour as the root node. Performing discrete optimization then basically consists of traversing a tree, heading for an optimal solution. One usually uses an *agenda* containing the tree nodes still to be visited. Search algorithms extract, visit and branch nodes to obtain possible successor nodes, which are appended to the agenda. The search terminates as soon as the agenda becomes empty. Examples are generic breadth-first search, where nodes are extracted in First-In-First-Out(FIFO) order, and depth-first search, employing Last-In-First-Out (LIFO) extraction order. Advanced approaches employ heuristic estimates to direct the search [48].

A common approach to parallelize discrete optimization is to distribute unvisited nodes over the set of available processors. For that purpose, we implemented the traversal of the subtree mounted at an unvisited node as a Cohesion task. Consequently, the task pool effectively is equivalent with the agenda.

6.3.2 Identifying crosscutting concerns

Aspect-oriented design [50] distinguishes between *core* and *crosscutting concerns*. One approach to identify crosscutting concerns is to dissect the system into *essence* and *incarnation* [49]. The essence of a system is constituted by its requirements and properties that would exist, even if there was perfect technology at hand. The incarnation of a system is the sum of all persons and machines realizing the essence. Both differ only because of the lack of perfect technology.

Load balancing is the first example of a crosscutting concern, whose necessity results from the inability of a single processor to solve the problem in acceptable time. Another one is fault tolerance, resulting from unreliability. The third crosscutting concern is termination detection which stems from the lack of complete information about the status of the distributed search.



```
public void traverse(Agenda agenda, Node root) {
    agenda.insert(root)
    while(!agenda.isEmpty()) {
        Node node =    agenda.extract()
        node.handle()
        agenda.insert   ( node.branch() )
    }
}
```

Fig. 9. Aspect-oriented decomposition of a heuristic search procedure.

6.3.3 Implementation

An *aspect* describes the modular implementation of the client-part of a crosscutting concern. This definition separates the crosscutting client-part and the conventionally implemented server-part and allows a modular and reusable implementation of the concern. Only the client-part has to be adapted to a new environment. For our implementation, we used *AspectJ* which provides the following concepts: *join point*, *pointcut*, *advice* and *introduction*. Introductions statically extend the signature of classes, e.g. by adding fields or methods. Pointcuts allow the declaration of sets of join points (well-defined points in the control flow), to which advice can be applied. For example, the pointcut `call(public void Agenda+.add(Node))` includes all calls adding nodes to objects implementing the interface `Agenda`. As explained subsequently, all functionality required for integration can be coherently encapsulated within AspectJ aspects. Existing code implementing search remains unmodified (cf. Figure 9).

Load Balancing. To achieve dynamic distribution of unvisisted tree nodes, COHESION's load balancing module requires certain information — for example the current load status of the local agenda. To provide this information, we define pointcuts intercepting calls changing this status. Advice attached to these pointcuts calls observing methods of the corresponding COHESION module. These pointcuts combined with the appropriate advice represent the client-part, while the original load balancing module acts as the server-part (see Figure 9).

Also, methodology is required to balance the load status of agendas on different hosts, e.g. to initiate the transfer of tree nodes representing their corresponding subtrees from one agenda to its remote counterpart. Thus, agendas must provide the required signature that is accessed by the instrumented COHESION module. Aspect-orientation supports this by the concept of introduction. The agenda's signature is enriched by a method to steal tasks, like `public Node[] Agenda.stealNodes() { /* ... */ }`.

Fault tolerance. Here, the corresponding COHESION module is interested in actions transferring tree nodes between hosts. By monitoring these, it keeps track of dispatched workload and all significant changes to the agenda.

		Runtime [s]	**Speedup**	**Efficiency [%]**
Sequential	Type-I	24678		
	Type-II	26120		
Parallel	Stable	562	43,9	54,9
	Volatile	1718	14,4	18,0

Table 2
Performance results of Traveling Salesperson Problem

This functionality is added through pointcuts intercepting balancing actions, whose advice in turn informs the observing Cohesion bundle. Our implementation creates proxies whenever a tree node is farmed out for remote traversal. When it decides, that a task should be restarted, i.e. failure detection signalizes the loss of the host the node was sent to, a local copy of the node is reinserted into the local agenda. For dealing with CPU volatility the affected host tries to send back partially handled workload, i.e. the remaining unvisited nodes of the subtree received through its root node.

Termination Detection. We use a tree-based approach for termination detection. Unvisited nodes are counted and hosts are informed about the remote traversal of delegated subtrees. To be able to do this, the respective module must be informed every time a node has been visited or is branched (the corresponding pointcuts are illustrated in Figure 9). Also, it must be notified every time a node is moved to another host through load balancing and when a node and it's subtree have been traversed remotely. This is implemented by pointcuts that intercept balancing actions.

6.3.4 Performance Evaluation

We conducted tests with random instances of the *Traveling Salesperson Problem* (TSP) solved by a *Branch-and-Bound* approach. Bounds are piggybacked on balancing and traversal notification messages. This optimization turned out to be more efficient than groupcasting them. The testbed consisted of 48 Cohesion nodes running on Type-I and 32 Cohesion nodes running on Type-II nodes. Table 2 shows results based on 10 program runs for each setting. The simulation of volatility is based on per node event intervals of 5 minutes with 10% host failures. Speedups and efficiencies are computed using the sequential runtime on the faster host type (Type-I). The results show, that even under worst-case volatility, our Desktop Grid approach can be beneficially employed for speeding-up real world applications.

7 Conclusion

In this paper, we reported on the design and application of our Desktop Grid platform Cohesion. Our research aims at extending the scope of Desktop Grid computing beyond plain master/worker parallelism applied by classical Desktop Grid approaches. In the light of a class of (irregularly structured) task-parallel problems we discussed, how P2P principles can be employed to realize parallel programming models with more complex interaction patterns.

An important finding of our work is, that employing P2P concepts results in a multitude of options for realizing pertinent system functionality (e.g. network substrates, multicast operations, or membership protocols). However, application performance often strongly depends on choosing a specific point within this design space, which is also dependent on the actual parallel environment. Hence, Cohesion is based on a microkernel system architecture that allows such application specific customization of system functionality in a robust, coordinated, and secure way.

Still, in spite of all the factors explored in this paper, this research is not an exhaustive evaluation of all aspects of P2P Desktop Grid computing. In particular, it is not yet clear, whether the combination of the Desktop Grid approach and P2P techniques can be used to open up additional problem classes. An interesting question is, whether data-parallel problems can also profit from the Desktop Grid approach. Dealing with volatility would require frequent repartitioning and remapping of the data set. In principle such operations can also be efficiently accomplished by P2P style communication. However, due to their synchronous nature their performance impact can be expected to be far more relevant to the overall performance.

References

[1] Cohesion website, http://www.cohesion.de, last accessed 06/05/2007.

[2] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, 2nd Edition, Morgan Kaufmann, 2004.

[3] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organizations, The International Journal of High Performance Computing Applications 15 (3) (2001) 200–222.

[4] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, A. A. Chien, Characterizing and evaluating desktop grids: An empirical study, in: Proc. of International Parallel and Distributed Processing Symposium, Sante Fe, New Mexico, 2004.

[5] D. P. Anderson, G. Fedak, The computational and storage potential of volunteer computing, in: Proc. of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), Singapore, 2006, pp. 73–80.

[6] I. Foster, A. Iamnitchi, On death, taxes, and the convergence of peer-to-peer and grid computing, in: International workshop on peer-to-peer systems (IPTPS 2003), Berkeley, CA, USA, 2003.

[7] I. Foster, Globus toolkit version 4: Software for service-oriented systems, Journal of Computer Science and Technology 21 (4) (2006) 513–520.

[8] I. Foster, C. Kesselman, J. Nick, S. Tuecke, The physiology of the grid: An open grid services architecture for distributed systems integration, available from http://www.globus.org, last accessed 03/21/2007.

[9] R. Gupta, V. Sekhri, A. K. Somani, CompuP2P: An architecture for internet computing using peer-to-peer networks, IEEE Transactions on Parallel and Distributed Systems 17 (11) (2006) 1306–1320.

[10] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage, in: 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA, 2004.

[11] A. Chien, B. Calder, S. Elbert, K. Bhatia, Entropia: architecture and performance of an enterprise desktop grid system, Journal of Parallel Distributed Computing 63 (2003) 597–610.

[12] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, O. Lodygensky, Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid, Future Generation Computer Systems 21 (3) (2005) 417–437.

[13] J. Verbeke, N. Nadgir, G. Ruetsch, I. Sharapov, Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment, in: Proceedings of the Third International Workshop on Grid Computing (GRID '02), Springer-Verlag, London, UK, 2002, pp. 1–12.

[14] Y. T. Kazuyuki Shudo, S. Sekiguchi, P3: P2P-based middleware enabling transfer and aggregation of computational resources, in: Proc. Cluster Computing and Grid 2005 (Fifth Int'l Workshop on Global and Peer-to-Peer Computing), Cardiff, UK, 2005.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, in: Proceedings of the ACM SIGCOMM '01 Conference, San Diego, California, 2001.

[16] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, V. Sunderam, Towards self-organizing distributed computing frameworks: The H2O approach, Parallel Processing Letters 13 (2) (2003) 273–290.

[17] M. Migliardi, V. Sunderam, The harness metacomputing framework, in: In Proc. of the Ninth SIAM Conference on Parallel Processing for Scientic Computing, S. Antonio (TX), USA, 1999.

[18] C. Engelmann, G. A. Geist, A lightweight kernel for the harness metacomputing framework, in: IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 1, IEEE Computer Society, Washington, DC, USA, 2005, p. 120.2.

[19] D. Kurzyniec, T. Wrzosek, V. Sunderam, A. Sominski, Rmix: A multiprotocol rmi framework for java, in: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society, Washington, DC, USA, 2003, p. 140.

[20] Y. Sun, C.-L. Wang, Solving irregularly structured problems based on distributed object model, Parallel Computing 29 (11-12) (2003) 1539–1562.

[21] W. Blochinger, W. Westje, W. Küchlin, S. Wedeniwski, ZetaSAT – Boolean satisfiability solving on desktop grids, in: Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005), Vol. 2, Cardiff, UK, 2005, pp. 1079–1086.

[22] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu, Peer-to-Peer Computing, Tech. rep., Hewlett Packard (2002).

[23] B. Ford, D. Kegel, P. Srisuresh, Peer-to-peer communication across network address translators, in: Proceedings of the 2005 USENIX Technical Conference, 2005.

[24] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, H. Casanova, On resource volatility in enterprise desktop grids, in: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, IEEE Computer Society, Washington, DC, USA, 2006, p. 78.

[25] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson, A new major SETI project based on Project Serendip data and 100,000 personal computers, in: Proc. of the 5th International Conference on Bioastronomy, 1997.

[26] S. Schulz, W. Blochinger, An integrated approach for managing peer-to-peer desktop grid systems, in: Proc. of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), Rio de Janeiro, Brazil, 2007, pp. 233–240.

[27] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture - A System of Patterns, John Wiley and Sons Ltd, 1996, chichester, UK.

[28] OSGi$^{\text{TM}}$- The Dynamic Module System for Java$^{\text{TM}}$, http://www.osgi.org, last accessed 03/21/2007.

[29] B. Calder, A. A. Chien, J. Wang, D. Yang, The entropia virtual machine for desktop grids, in: VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, ACM Press, New York, NY, USA, 2005, pp. 186–196.

[30] G. Antoniu, P. Hatcher, M. Jan, D. A. Noblet, Performance evaluation of jxta communication layers, in: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05), IEEE Computer Society, Washington, DC, USA, 2005, pp. 251–258.

[31] E. Halepovic, R. Deters, The costs of using jxta, in: Proceedings of the 3rd International Conference on Peer-to-Peer Computing, IEEE Computer Society, Washington, DC, USA, 2003, p. 160.

[32] J.-M. Seigneur, JXTA Pipe Performance, available from http://bench.jxta.org, last accessed 03/07/2007.

[33] K. Burbeck, D. Garpe, S. Nadjm-Tehrani, Scale-up and performance studies of three agent platforms, in: IEEE International Conference on Performance, Computing, and Communications, 2004, pp. 857– 863.

[34] Jive Software, Scalability: Turn it to eleven, available from http://www.igniterealtime.org, last accessed 03/07/2007.

[35] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.-C. Hugly, E. Pouyoul, B. Yeager, Project JXTA 2.0 Super-Peer Virtual Network, Tech. rep., Sun Microsystems (May 2003).

[36] XMPP Standards Foundation Website, http://www.xmpp.org, last accessed 03/21/2007.

[37] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[38] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky, Bimodal Multicast, ACM Trans. Comput. Syst. 17 (2) (1999) 41–88.

[39] XQuery 1.0: An XML Query Language, W3C Working Draft. Available from http://www.w3.org/TR/xquery, last accessed 03/21/2007.

[40] I. Gupta, T. D. Chandra, G. S. Goldszmidt, On Scalable and Efficient Distributed Failure Detectors, in: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, ACM Press, 2001, pp. 170–179.

[41] A. J. Ganesh, A.-M. Kermarrec, L. Massouli, Peer-to-Peer Membership Management for Gossip-based Protocols, IEEE Trans. Comput. 52 (2) (2003) 139–149.

[42] A. Das, I. Gupta, A. Motivala, SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol, in: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, IEEE Computer Society, 2002, pp. 303–312.

[43] T.-H. Lai, S. Sahni, Anomalies in parallel branch-and-bound algorithms, Communications of the ACM 27 (6) (1984) 594–602.

[44] K. H. Randall, Cilk: Efficient multithreaded computing, Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science (Jun. 1998).

[45] W. Blochinger, W. Küchlin, The design of an API for strict multithreading in C++, in: H. Kosch, L. Böszörményi, H. Hellwagner (Eds.), Proc. of 9th Intl. Conf. Euro-Par 2003, no. 2790 in LNCS, Springer-Verlag, Klagenfurt, Austria, 2003, pp. 722–731.

[46] B. Wilkinson, M. Allen, Parallel Programming. Techniques and Applications using Networked Workstations and Parallel Computers, Prentice Hall, New Jersey, 1999.

[47] W. Blochinger, C. Dangelmayr, S. Schulz, Aspect-oriented parallel discrete optimization on the Cohesion desktop grid platform, in: Proc. of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), Singapore, 2006, pp. 49–56.

[48] A. Grama, V. Kumar, State of the art in parallel search techniques for discrete optimization problems, IEEE Transactions on Knowledge and Data Engineering 11 (1) (1999) 28–35.

[49] S. M. McMenamin, J. F. Palmer, Essential systems analysis, Yourdon Press, Upper Saddle River, NJ, USA, 1984.

[50] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Co., 2003.