Capability-Aware Information Aggregation in Peer-to-Peer Grids

Methods, Architecture, and Implementation

Sven Schulz, Wolfgang Blochinger and Hannes Hannak

Journal of Grid Computing manuscript No.
(will be inserted by the editor)

Capability-Aware Information Aggregation in Peer-to-Peer Grids

Methods, Architecture, and Implementation

Sven Schulz · Wolfgang Blochinger · Hannes Hannak

Received: date / Accepted: date

Abstract Information aggregation is the process of summarizing information across the nodes of a distributed system. We present a hierarchical information aggregation system tailored for Peer-to-Peer Grids which typically exhibit a high degree of volatility and heterogeneity of resources. Aggregation is performed in a scalable yet efficient way by merging data along the edges of a logical self-healing tree with each inner node providing a summary view of the information delivered by the nodes of the corresponding subtree. We describe different tree management methods suitable for high-efficiency and high-scalability scenarios that take host capability and stability diversity into account to attenuate the impact of slow and/or unstable hosts. We propose an architecture covering all three phases of the aggregation process: Data gathering through a highly extensible sensing framework, data aggregation using reusable, fully isolated reduction networks, and application-sensitive data delivery using a broad range of propagation strategies. Our solution combines the advantages of approaches based on Distributed Hash Tables (DHTs) (i.e., load balancing and self-maintenance) and hierarchical approaches (i.e., respecting administrative boundaries and resource limitations). Our approach is integrated into our Peer-to-Peer Grid platform COHESION. We substantiate its effectiveness through performance measurements and demonstrate its applicability through a graphical monitoring solution leveraging our aggregation system.

Keywords Hierarchical Information Aggregation · Peer-to-Peer Systems · System Architecture · Networked System Monitoring

Sven Schulz
University of Tuebingen
Tel.: +49-7071-2970472
Fax: +49-7071-295160
E-mail: schulzs@informatik.uni-tuebingen.de

Wolfgang Blochinger
University of Tuebingen
Tel.: +49-7071-2970469
Fax: +49-7071-295160
E-mail: blochinger@informatik.uni-tuebingen.de

Hannes Hannak
University of Tuebingen
E-mail: hannak@informatik.uni-tuebingen.de

1 Introduction

Peer-to-Peer (P2P) Grids pool unused resources of non-dedicated workstations for tackling computationally demanding problems. Specifically, they leverage P2P principles to enable complex interaction patterns among participating hosts and thus efficiently support advanced parallel programming models. However, P2P Grids differ significantly from traditional high performance systems. Particularly, resources exhibit a high degree of volatility and heterogeneity: Depending on the usage-patterns of the participating hosts, considerably heterogeneous resources join and leave the grid in a frequent and unpredictable manner. These characteristics pose enormous challenges to system as well as application designers.

Today, large P2P Grids, in principal, can realize supercomputer-level performance: Since 2003, there has been a stable growth in the number of registered DNS names to over half a billion entries [1]. As many hosts are part of private networks, the actual number of computers in use can be expected to be even higher. In the same time frame, the performance of state-of-the-art CPUs has significantly increased. However, in order to be able to effectively exploit this plethora of computational power, further development and adaptation of key techniques in distributed and high performance systems become necessary.

In this work we investigate on efficient information aggregation techniques for P2P Grids. Information aggregation is the process of summarizing information across the nodes of a distributed system. For large-scale systems it is inevitable to perform information aggregation in a hierarchical manner as exposing all information to all nodes would quickly overwhelm even the most powerful nodes. Specifically, hierarchical approaches based on aggregation trees can provide information with different levels of detail by progressively summarizing information along the edges of a tree spanning all nodes.

Fundamental distributed paradigms and algorithms are based on information aggregation [2], including leader election, voting, service and resource placement, multicast tree formation, and error recovery. Thus an information aggregation service can serve as a basic building block for the design of sophisticated P2P Grid components. Despite of its importance, there are numerous open challenges in design and implementation of aggregation systems that are considered worthy of future research [3]. Especially, the cost of reconfigurations caused by high node volatility can become significant if the variance in performance and stability features of nodes are not taken into account. This aspect is of particular importance when the considered networked system is a P2P Grid.

In this paper we present an information aggregation system especially tailored for P2P Grids. Specifically, we make the following contributions:

1. We describe methods for both highly efficient aggregation tree management for small-scale groups and scalable aggregation tree management for large-scale groups. Both methods have only minimal requirements on the underlying system platform and can be extended to take the capability heterogeneity into account, that is predominant in P2P systems. We demonstrate that it is indispensable for efficient aggregation tree maintenance to account for node stability and describe how our capability-aware approach fulfills this requirement.
2. We describe a flexible aggregation architecture that tightly integrates with the modularization approaches of state-of-the-art distributed system architectures and allows for customizing each phase of the aggregation process in a very flexible application-controlled way. We integrated our information aggregation system into our modular P2P Grid platform COHESION [4] and present results of a comprehensive experimental performance evaluation.

3. We propose to use a dedicated measurement scheduler based on resource usage quotas and lottery scheduling to impose soft limits on resource consumption (CPU time, network bandwidth, and secondary storage capacity) by the information aggregation subsystem.

The code of COHESION as well as the code of the presented information aggregation system can be downloaded from `http://www.cohesion.de`.

1.1 Paper Organization

The remainder of this paper is organized as follows: In Section 2, we first discuss alternative approaches to information aggregation in large-scale distributed systems. Then, we give a brief account of P2P Grid Computing and our P2P Grid platform COHESION in Section 3. In Section 4, we identify the requirements of information aggregation in general and those that are specific to P2P Grid systems. In Section 5, we give a top-level description of our approach to information aggregation. Section 6 describes in detail our techniques for constructing and maintaining a virtual tree topology on the set of nodes. These methods lie at the core of our approach, providing the structure for deploying a reduction network which performs the actual aggregation. Section 7, 8, and 9 explain in detail the subsystems of our aggregation system and its integration into COHESION. In Section 10, we present the evaluation of our system by performance experiments. Section 11 discusses the design of a graphical monitoring solution as an application of our information aggregation system. Finally, Section 12 summarizes our contributions.

2 Related Work

Current approaches to information aggregation in loosely coupled distributed systems are either based on flat, gossip style communication models with special termination/convergence properties or employ a tree overlay topology to hierarchically compute aggregate values.

In [5] an aggregation method based on an epidemic protocol is discussed. In this approach, every node periodically selects a peer at random, exchanges values denoting the system state, and performs an aggregation specific computation. Based on a basic protocol to compute averages, several other aggregation functions, like sum and variance, are realized. While epidemic protocols are known to be exceptionally robust, their efficiency is moderate only.

Tree topology based aggregation methods can be classified in static and dynamic approaches, depending on the way the tree topology is defined and maintained. Dynamic approaches either depend on an unstructured communication model or leverage structured network overlay technology. Subsequently, we discuss representative examples of these classes.

Ganglia [6] is a scalable distributed monitoring system mainly targeted at federations of clusters. It employs a listen/announce protocol based on multicast for monitoring individual clusters. Thus, all nodes of a cluster collect and store the state of all other nodes. Membership is maintained by a multicast heartbeat protocol. Within federated clusters, Ganglia uses a tree based protocol for information aggregation, where leaves of the tree are representative nodes of each cluster. For handling failures, multiple nodes of a single cluster can be specified as representatives. Aggregation at inner nodes of the tree is accomplished by periodically polling child nodes. Configuration files are used to specify the structure of the aggregation tree, which typically reflects the administrative topology. As Ganglia takes a static approach, it is not appropriate for highly dynamic P2P Grids.

Astrolabe [7] is a comprehensive approach for robust and scalable monitoring and management of distributed systems. For information aggregation it employs a tree structure which reflects the administrative organization of the distributed system. It is based on an unstructured gossip protocol for maintaining the tree topology and for information dissemination. All aggregate values of a subtree are replicated on every node of the subtree, such that all respective queries can be answered with local information. For specifying aggregation functions, Astrolabe uses a restricted form of mobile code based on SQL syntax.

The aggregation method of SDIMS [8] leverages the internal routing protocols of distributed hash tables to establish a tree based aggregation hierarchy. With this approach, the union of search paths for a key from different nodes forms a tree. As keys are based on attribute names, different attribute names are mapped to different trees, such that each node acts as an intermediate point of aggregation for some attributes. Thus, the onus of aggregation can be distributed among the participating hosts. In order to achieve administrative isolation, so called autonomous distributed hash tables (ADHT) are employed which ensure that search paths are always contained in the smallest possible domain and that search paths for a key from different nodes of a domain converge at a node part of that domain. On top of the ADHT layer, the aggregation management layer (AML) is responsible for maintaining attribute tuples, performing aggregations, and storing aggregated values. For increasing robustness in the case of topology reconfiguration, the AML layer performs replication in time (lazy and on-demand) and additionally replication in space.

Both Astrolabe and SDIMS do not take varying host capabilities and stability into account. Thus, the efficiency of the aggregation process is limited by the speed of slow hosts and the accuracy may be seriously impaired by unstable hosts.

Sensor networks share some of the characteristics of P2P Grids, like constrained resources, limited view of the whole system, and a high degree of volatility. However, heterogeneity is not a major issue in sensor networks. TAG [9] is an aggregation system especially addressing these sensor network specific issues. It supports declarative aggregation queries inspired by aggregation operator in an SQL style query syntax. The system represents an in-network aggregation approach and is based on routing trees. Trees are constructed in an ad-hoc fashion leveraging the range restricted broadcast capabilities of individual sensors.

3 Peer-to-Peer Grid Computing

In this section we first give a brief account of the field P2P Grid Computing. Subsequently, we give an overview of our P2P Grid Computing platform COHESION.

3.1 Characteristics of Peer-to-Peer Grids

P2P Grids belong to the class of Desktop Grids [10]. Traditional Desktop Grid systems are based on a client/server operational model. As a consequence, respective applications are most often based on trivial parallelism following the master/server or bag of tasks model. Prominent representatives of Desktop Grid platforms are BOINC [11] for volunteer computing and Entropia [12] for enterprise deployment scenarios.

The specific goal of the P2P Grid approach is to extend the applicability of Desktop Grid Computing towards non-trivial parallelism. The P2P principles enable complex interaction patterns among the participating hosts such that advanced parallel programming models can be realized. For example, parallel applications based on dynamic problem decomposition, like discrete optimization or constraint satisfaction, can benefit from the advanced capabilities of P2P Grids. Here new tasks are continuously generated at different locations and must be dynamically balanced over the available processors. Typical examples of P2P Grid systems are Personal Power Plant (P3) [13] and JNGI [14], both implemented on top of the JXTA P2P protocol suite.

In general, for P2P (and also Desktop) Grid Computing no hard quality of service guarantees are possible due to the high volatility and heterogeneity of resources. Enterprise scale deployment scenarios can probably realize soft guarantees by enforcing administrative guidelines for resource sharing (e.g., computers must join the grid after office hours). However, on the other hand, P2P Grids can be operated by virtually all institutions and can deliver considerable computational power at virtually no extra cost [15].

P2P Grids differ notably from other P2P based applications, like file sharing or instant messaging: In order to achieve high parallel efficiency, economical use of resources is of primary interest in P2P Grids. Resources are also limited due to constraints determined by the resource owners which are typically the users of the computers. Specifically, in P2P Grids, no user intervention can be assumed such that any kind of fault should be handled transparently.

Traditional Grids, realizing virtual organizations and P2P Grids ultimately pursue the same goal: aggregation of resources beyond local administrative domains. However, the two approaches face different requirements and constraints, like target communities (limited trust vs. no trust) or nature of resources (high-end vs. end-user) [16]. System architectures for building virtual organizations must specifically deal with interoperability issues, like standardization of protocols and interfaces. In contrast, architectures for constructing and operating P2P Grids must foremost reflect the high degree of resource volatility. Also, only little administrative overhead is acceptable, since typically no additional personnel is available for operating P2P Grid installations. As a consequence, lightweight, modular, and self-organizing system architectures become mandatory, since they reduce software and runtime complexity and can also adapt to the prevailing dynamism.

3.2 Cohesion Peer-to-Peer Grid Platform

COHESION overcomes the limitations of conventional Desktop Grid platforms by employing P2P style communication via interchangeable network substrates. However, this transition to P2P principles results in a plethora of design options on every layer of the system, that can no longer be satisfactorily handled by traditional monolithic architectures. COHESION addresses this issue by supporting extension and customization of all major system components through a system design based on an industrial-strength microkernel technology. For the purpose of this work, we limit our discussion to the relevant parts of the COHESION architecture which are our microkernel approach and the components of the virtualization layer (cf. Figure 1). An in-depth treatment of the system architecture can be found in [4, 17, 18].

3.2.1 Microkernel

The microkernel of COHESION is based on the dynamic module system provided as part of the *Open Services Gateway Interface* (OSGi) standard [19]. An application targeted for the

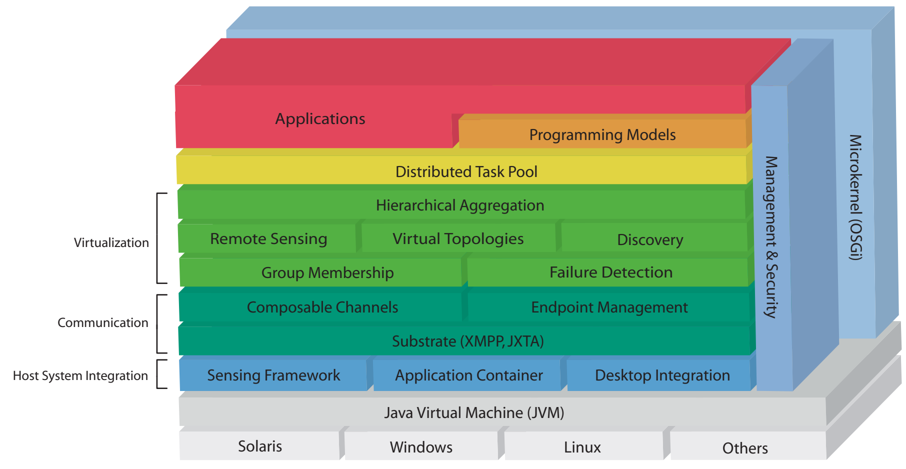


Fig. 1 Layered architecture of COHESION

COHESION platform consists of one or more modules, which are called *bundles* in OSGi jargon. Bundles are dynamic modules, thus they are first-class objects with a lifecycle, which can be deployed into and undeployed from a running OSGi framework instance. Due to this dynamism the preferred programming paradigm in COHESION is component-based programming. Components may use services from and provide services to other components. Services are *POJOs* (Plain Old Java Objects) that can represent virtually anything and are managed by a security-aware *service registry*. The lifecycle of a component, i.e., instantiation, binding of requested services, publishing of provided services and destruction, is managed by the OSGi *Declarative Services* framework. Hence, a bundle developer is relieved from the burden of driving the service logic and handling all the dynamism caused by emerging and vanishing services. We heavily use the service model to realize the extensibility and flexibility requirements (G4) identified in Section 4.

3.2.2 Virtualization

The virtualization layer establishes an abstraction between distributed resources and higher level services (including applications). On this layer, COHESION provides services to discover and organize resources, to monitor their availability, and to coordinate their usage.

A *group* is the central abstraction within the overall architecture of COHESION. Groups are used to organize the (initially unstructured) set of participating nodes into functional communities where nodes interact to achieve a common purpose. Technically, a group is a dynamic set of nodes defining a scope for isolated interaction (multicast communication). COHESION groups are hierarchical and hence are a capable tool to model the structure of an application domain. Nodes can be part of an arbitrary number of groups, but are at least a member of the root group.

A (group) *membership protocol* (GMP) enables nodes to agree on a common view of a group's membership. However, this agreement does not necessarily imply that all nodes actually share the same view. A GMP at least has to handle join and leave requests of nodes. Note that the way in which a resulting change in membership is reflected by local mem-

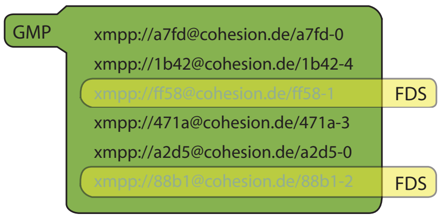

GMP
xmpp://a7fd@cohesion.de/a7fd-0
xmpp://1b42@cohesion.de/1b42-4
xmpp://ff58@cohesion.de/ff58-1
FDS
xmpp://471a@cohesion.de/471a-3
xmpp://a2d5@cohesion.de/a2d5-0
xmpp://88b1@cohesion.de/88b1-2
FDS

Fig. 2 Combined membership view resulting from masking out nodes considered as faulty by the FDS.

bership lists is protocol dependent. Existing protocols differ in many aspects including distribution and coverage of membership lists, resulting in different characteristics concerning view convergence and scalability. COHESION provides two different GMP implementations: one that utilizes mechanisms of the underlying substrate (JXTA discovery service or XMPP Multi User Chats) with complete views and a substrate-agnostic fully decentralized, self-organizing membership protocol with partial views of size $O(log(n))$ based on SCAMP [20].

If a node crashes or becomes isolated by faulty networks links, the node's membership can not be canceled through explicit unsubscription by the GMP. Thus, COHESION uses a dedicated failure detection service (FDS) to detect such ungraceful node departures. We use a fully decentralized P2P periodic randomized probing protocol called SWIM [21]. Since nodes that are wrongly reported as faulty (false positives) can have serious impact on protocols on higher layers of the system, SWIM employs indirect probing and suspicions to reduce their rate.

Both mechanisms, membership management and failure detection, are used to compute the set of available nodes within a group by masking out those nodes from the GMP managed view that are considered faulty by the failure detection service (see Figure 2).

4 Requirement Analysis

As stated in the introduction, many fundamental problems in distributed computing can be solved using information aggregation. To serve as such a generic building block for P2P Grid Computing services, an information aggregation system must satisfy a number of requirements.

General requirements for information aggregation systems designed for large-scale distributed systems [7, 8] include:

(G1) Scalability. Typically, there is a large number of participating nodes and also a large number of system attributes to keep track of. Thus, the system must be explicitly designed to be scalable by not relying on global knowledge or central control. For example, a flat aggregation scheme would simply collect information from all nodes on a single node, calculate a summary value, and broadcast the result to all participating nodes. However, such an approach turns out to be of limited scalability and robustness. In contrast, a hierarchical aggregation scheme distributes the onus of aggregation over all participating nodes. Rather than exposing all information to all nodes, hierarchical aggregation allows a node to access detailed view of nearby information and summary views of global information. Particularly, systems that aggregate information through reduction trees allow nodes to access information they care about while maintaining scalability.

(G2) Efficiency. The system should use the aggregation infrastructure effectively. Since the same attributes may be monitored by multiple parties simultaneously, it is indispensable that the whole infrastructure (i.e., sensors, aggregation trees, and delivery channels) is shared. Additionally, to avoid unnecessary waste of resources, one needs to prevent duplication of measurements.

(G3) Robustness. Faults are frequent in large-scale distributed systems, thus the aggregation system must be robust and should tolerate node failures and link failures. It should also adapt to the resulting changes in network composition and topology.

(G4) Flexibility/Extensibility. Since application requirements are diverse, the system should support a broad range of both, sensors providing system state, as well as aggregation and result propagation strategies. The latter includes application-level control over how and when sensor data should be gathered, transmitted, aggregated, and distributed. Policy control should be implemented in a way that tightly integrates with the modular approach of state-of-the-art distributed system architectures [4,22]. Where the available functionality is not sufficient or inappropriate, the system must allow for extension and customization through the deployment of additional sensors or result propagation strategies.

(G5) Hierarchical Addressing. As explained in (G1) an aggregation system must be hierarchical to scale. This implies that the system exposes various levels of aggregation. Hence, a flexible, location-independent addressing scheme that reflects the hierarchical organization of the aggregation system is needed.

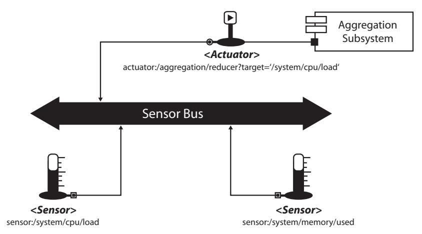
In addition to these general requirements of information aggregation there is a number of unique challenges when the target system is a P2P Grid.

(S1) Volatility. Resources in P2P Grids are not available permanently [4,23,10,24]. The *volatility* of a system is the fluctuation of the overall resource availability within the system. In P2P Grids volatility is much more pronounced than in other kinds of parallel or distributed systems like compute clusters. This can be attributed to reduced system isolation and typically lesser reliability of host components (like lacking redundancy), resulting in a multiplication of possible error sources. Furthermore, non-dedication, i.e., the nodes of a P2P Grid are primarily used for other purposes, considerably contributes to the observed increase in volatility. The resulting requirement is closely related to robustness (G3). However, the impact of regular fluctuation in resource availability is typically much more pronounced than that resulting from failure. Hence, the system must not only handle occasional error conditions but must be tailored to cope with constant flux.

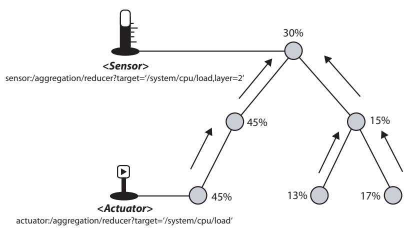
(S2) Resource Limitations. Many nodes in a P2P Grid are of limited capability since resources like CPU time, network bandwidth, and secondary storage capacity are constrained by the owner of the resources. Thus, the share of these resources consumed by the aggregation system must be bounded to avoid interference with the actual main purpose of the system.

(S3) Capability Heterogeneity. The nodes of a P2P Grid are most typically heterogeneous in terms of capability. Hence, the strategy of distributing the aggregation onus uniformly across participating nodes (commonly used in networks of nodes that are on par concerning capability) is not applicable.

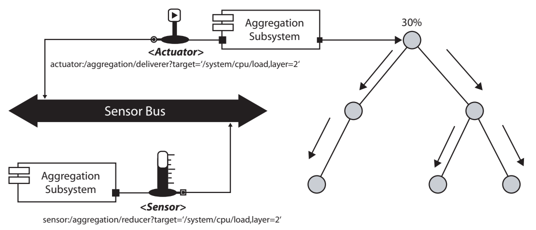
(S4) Administrative Segmentation. In contrast to cluster and supercomputer environments, P2P Grids typically span several administrative domains. This has two immediate conse-


Aggregation Subsystem
<Actuator>
actuator:/aggregation/reducer?target='/system/cpu/load'
Sensor Bus
<Sensor>
sensor:/system/cpu/load
<Sensor>
sensor:/system/memory/used

(a) **Phase I**: *Information gathering*. Sensor values are delivered over a sensor bus to the aggregation subsystem. The set of supported metrics can be easily extended by deploying custom sensors. A measurement scheduler prevents measurement duplication and enforces resource limits configured by the host owner.


30%
<Sensor>
sensor:/aggregation/reducer?target='/system/cpu/load,layer=2'
45%
15%
45%
13%
17%
<Actuator>
actuator:/aggregation/reducer?target='/system/cpu/load'

(b) **Phase II**: *Information aggregation*. Sensor values from hosts within the aggregation group are summarized along the edges of an aggregation tree providing increasingly condensed information toward the root node. Reducers performing the evaluation of the aggregation function again expose aggregate values through dedicated sensors.


Aggregation Subsystem
<Actuator>
actuator:/aggregation/deliverer?target='/system/cpu/load,layer=2'
30%
Sensor Bus
Aggregation Subsystem
<Sensor>
sensor:/aggregation/reducer?target='/system/cpu/load,layer=2'

(c) **Phase III**: *Delivery of aggregated values*. Aggregate values are fetched by an actuator that e.g., reuses the aggregation tree to efficiently deliver the aggregate values to a subset or all of the nodes in the aggregation group.

Fig. 3 Phases and Components of the information aggregation process.

quences: First, these domains are typically isolated by firewalls and address translating devices (NAT). Such network segmentations must be bridged by the system. However, intersegment traffic should be limited to avoid bottlenecks. Second, confidential sensor data must be protected from unauthorized access. Thus, the system should be able to process queries concerning information about nodes that are part of a specific administrative domain completely within that domain. Additionally, the system must exhibit fine-grained control over which attributes are exposed to others.

Note that these requirements affect either overall system architecture (G2, G4, S2, and S4), tree management methods (G1, G3, G5, and S3), or both (S1). We discuss how our solution accounts for them in Section 5 and Section 6 respectively.

5 Information Aggregation Architecture

In this section we give a brief overview of the general architecture of our information aggregation system. Detailed discussions on each aspect of the system can be found in the subsequent sections.

The process of information aggregation can be decomposed into three phases (see Figure 3): data gathering, data aggregation, and distribution of the aggregated data.

Data gathering within our solution is addressed by providing a highly extensible sensing framework (see Section 7). The central abstraction is a *sensor bus* that leverages the microkernel design of COHESION to allow modules to create and deploy custom sensors to capture system state (see Figure 3a). The sensor bus actively schedules measurements using a dedicated measurement scheduler, thus enforcing resource limitations configured by the host owner (S2) and preventing duplication of measurements (G2).

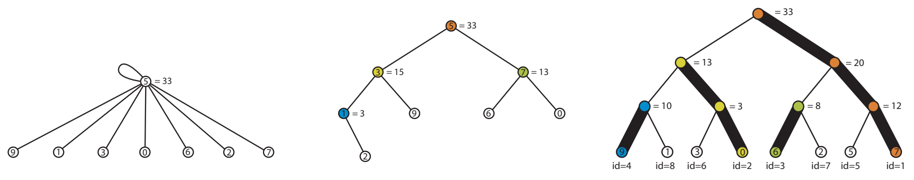
Aggregation is done in an efficient and scalable way along the edges of a self-healing logical *aggregation tree* (see Figure 3b and Section 9 for a detailed description). The aggregation tree spans all hosts within a logical partition of the whole host set. These logical partitions are called *aggregation groups* and are realized as COHESION groups, which fosters isolation and ensures connectivity. The aggregation tree consists of *reducers* collecting values from its children and providing aggregated values to their parent. To use the aggregation infrastructure most efficiently (G2), we maintain a single shared reducer network per sensor that is used to satisfy an arbitrary number of parallel aggregation requests and typically a single aggregation tree per application that is shared among all reducer networks.

Tree management is designed to be highly customizable. Various providers for performing tree management operations can be plugged-in to realize different tree management strategies. We use this feature to realize different strategies for prototypical use cases: highly-performant and highly-scalable groups. Furthermore, we incorporate mechanisms to reduce the impact of volatility by considering specific capabilities of hosts (performance, stability, or quality of network connection) when assigning a position within the reduction network. Thus, our solution is able to cope with both the diversity in requirements and the enormous dynamism (S1, G3) and heterogeneity (S3) prevalent in P2P grids.

Finally, delivery of sensor/reducer data within the reduction network is accomplished by allowing to probe remote sensors across network segmentations in a fully transparent secure fashion (S4). This is achieved by leveraging the overlay network provided by the COHESION platform (see Section 7). Delivery of aggregate values (see Figure 3c) can be accomplished in an application sensitive way using a broad spectrum of strategies, including propagation along the aggregation tree or by using a groupcast protocol provided by the platform. Because of this flexibility, we can support a wide variety of applications with diverse requirements (G4).

6 Methods

Naively, information aggregation could be done by having a particular host fetch values from all other hosts in the system (see Figure 4a). However, such a centralized approach would violate the scalability requirement (G1), since a single host would have to handle a vast number of messages. Hence, aggregation is performed by merging data along the edges of a spanning tree which covers all the hosts in the system (see Figure 4b). While this approach distributes the onus of aggregation over a larger number of hosts, the indefiniteness of spanning tree construction makes consistent addressing (G5), i.e., specifying the set of hosts the aggregate value should be computed from, a challenging task. To circumvent this limitation, we impose a logical overlay on the set of hosts instead of using the spanning tree directly [8]. The overlay topology is a tree, where each host in the system is a leaf node (see Figure 4c). Upper levels of the tree are populated by having selected hosts simulate additional nodes. As these nodes have no direct physical counterpart, we call them *virtual*



(a) Centralized aggregation.

(b) Spanning tree covering all hosts in the system.

(c) Logical tree overlay. Nodes connected by thick lines represent V-Nodes collocated on the same host.

Fig. 4 Topologies used by different aggregation methods. While the centralized approach (a) does not scale and plain tree-based (b) methods do not allow for consistent addressing, adopting a logical tree overlay based on V-Nodes (c) ensures that both requirements are satisfied.

Method	**Requirements**	**Features**		
	View Coverage	Scalability	Efficiency	Capability Support
ID Order	Complete	Low	**High**	Static*
Competition	Partial	**High**	Low	Dynamic

Table 1 Feature/Requirement comparison for the implemented schemes (* see Discussion in Section 6.1.1).

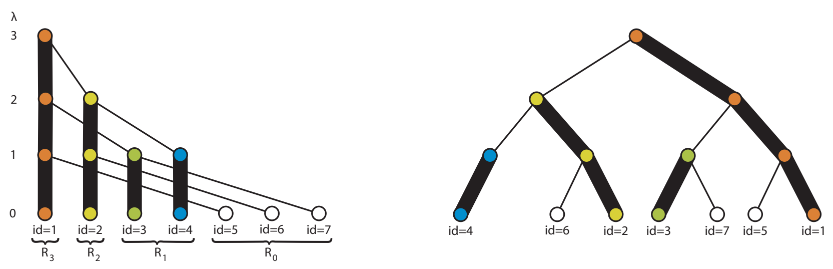
nodes or more shortly *V-Nodes*. Leaf nodes provide initial input to the aggregation tree. Each V-Node u performs aggregation locally by applying an *aggregation function*

$$f(u) := f(\{v \in children(u)\})$$

to the values provided by its child nodes. A *level-λ aggregate*, where $\lambda \in \{0, \lambda_{max}\}$ is the value resulting from performing the aggregation process up to level λ. While a level-0 aggregate is the raw input value provided by one of the hosts, a level-λ_{max} aggregate summarizes information from all hosts in the system. In Figure 4c there are four level-1 aggregates (10,3,8,12), two level-2 aggregates (13,20) and one level-3 aggregate (33). Since aggregates on the same level are characterized by the same level of detail, the tree overlay allows for consistent addressing: An *aggregation point* is given by a pair $(host, \lambda)$ and is resolved by following λ parent links starting from the given host. While $(4,2)$ and $(2,2)$ reference the same aggregation point, $(7,2)$ references another aggregation point that exposes the same level of summarization, i.e., both aggregation points summarize information from four hosts.

An appropriate selection of hosts to simulate V-Nodes is critical for the overall performance of the aggregation system for two reasons: First, processing incoming values from lower levels, evaluating the aggregation function, and communicating aggregated values to higher levels consumes resources. Placing more than a single V-Node on a host may exceed resource-constraints. Second, the cost of reconfiguration caused by a vanishing host increases with the number of hosted V-Nodes. Hence, it is preferable to have the most stable hosts simulate V-Nodes on the highest levels. Therefore, our approach is to prefer hosts that are more capable in terms of both performance and stability. We call this feature of a tree management method *capability awareness*.

In the following sections, we describe two different tree management methods with different requirements and features (see Table 1) and describe how capability awareness can be incorporated.



(a) Hosts sorted by maximum V-Node layer λ.

(b) V-Nodes laid out as logical aggregation tree with no edge crossings.

Fig. 5 V-Node allocation and link associations for an aggregation group with seven hosts in the ID-based topology management scheme.

6.1 ID-based Method

As discussed in Section 3, COHESION is a representative of a new class of P2P Grid Computing platforms supporting applications beyond trivial parallelism. As these applications typically require inter-processor communication, they are unlikely to scale to hundreds of thousands of processors. Instead, the main focus shifts to maximizing efficiency. As efficiency heavily depends on an up-to-date view of the available compute resources contributed by participating hosts, it is indispensable to quickly and accurately detect host arrivals and departures. Consequently, COHESION employs a group model with complete membership lists where each group member knows all other group members. The idea of the ID-based tree management method is to use already available information on group membership to construct and maintain a binary aggregation tree.

The levels of the aggregation tree are populated by selecting hosts according to a total order R imposed on the set of hosts within the aggregation group. Obviously, the whole structure of the binary aggregation tree can be derived from evaluating R for a given set of hosts that are member of the aggregation group (cf. Figure 5). Let $pos_R(u)$ be the position of host u in the aggregation group G with respect to R. Then host u is simulating V-Nodes up to level λ, if

$$pos_R(u) \in R_\lambda := \begin{cases} \left] \left\lceil \frac{|G|}{2^{\lambda+1}} \right\rceil, \left\lceil \frac{|G|}{2^{\lambda}} \right\rceil \right] \subset \mathbb{N} & for \quad \lambda \in [0, \lceil log_2(|G|) \rceil [\\ \{1\} & for \quad \lambda = \lceil log_2(|G|) \rceil \end{cases} \tag{1}$$

Additionally, a level-λ V-Node $\tilde{u}_\lambda$ located at host u is connected to the collocated V-Node $\tilde{u}_{\lambda-1}$ and to the V-Node $\tilde{v}_{\lambda-1}$ located at host v with

$$pos_R(v) = pos_R(u) + \left\lceil \frac{|G|}{2^\lambda} \right\rceil, \tag{2}$$

if it exists.

The topology management algorithm executed by each host is as follows: After becoming a member of the aggregation group, the host u is provided with a dynamic view of the hosts within the group, including itself. As full knowledge about the group members is necessary to compute $pos_R(u)$, we must employ a group model with complete views (cf. Section 3.2). Whenever this view is updated due to the arrival or departure of a host, $pos_R(u)$

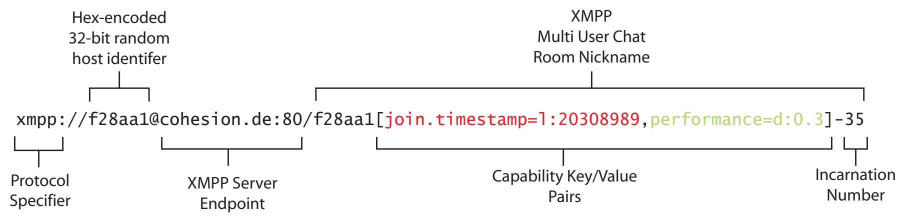

Hex-encoded 32-bit random host identifer
XMPP Multi User Chat Room Nickname
xmpp://f28aa1@cohesion.de:80/f28aa1[join.timestamp=l:20308989,performance=d:0.3]-35
Protocol Specifier
XMPP Server Endpoint
Capability Key/Value Pairs
Incarnation Number

Fig. 6 Capabilities (join timestamp and relative performance) used to evaluate the joint capability function c_j encoded in a COHESION host identifier.

is recalculated. Comparison with the former value $pos_{R^*}(u)$ produced on the last update results in creation $(pos_R(u) > pos_{R^*}(u))$ or destruction $(pos_R(u) < pos_{R^*}(u))$ of local V-Nodes on the respective layers. Subsequently, links are created according to the rule given above. Note, that this scheme is highly efficient (G2) as no communication other than that for group membership management is required. However, scalability (G1) is limited by the necessity for a group model with complete membership lists. Hence, aggregation using ID-based tree management is perfectly suited for applications, where efficiency is of prime interest (e.g., as part of a distributed computation).

Although the method described here can be generalized easily to k-ary trees, we restrict our treatment to binary trees for simplicity. Note, that the aggregation function is evaluated $\frac{|G|-1}{k-1}$ times with k arguments for k-ary trees in an aggregation group G with $|G|$ members. Thus, increasing k trades evaluation complexity against invocation count. If we choose the host of a V-Node to be one of the hosts in the subtree rooted at the respective V-Node, there is a path from the V-Node to the hosting leaf V-Node which consists exclusively of intra-host links. Thus, with the assumption that intra-host communication is cheap, there is no reason to expect significant overhead for trees with small degrees.

6.1.1 Capability Awareness

In the case of capability agnostic topology management, the order R is defined by the lexical order of the unique host identifiers that are provided by the COHESION group membership subsystem. Making this management scheme capability aware is as simple as replacing the random order with one that reflects the capability of hosts. Therefore, we define R as the ascending value order of the *joint capability* values $c_j(u)$ of the hosts $u \in G$. We use the simple joint capability function

$$c_j(u) = p(u)\,a(u) \tag{3}$$

where $p(u)$ is the relative performance and $a(u)$ is the availability ratio of host u. The relative performance is computed using the comprehensive CPU2006 benchmark result database published by the *Standard Performance Evaluation Corporation* (SPEC) [25] as the quotient of the benchmark result of the host $spec\_cpu(u)$ compared to the benchmark result of a reference machine $spec\_cpu_{ref}$

$$p(u) = \frac{spec\_cpu(u)}{spec\_cpu_{ref}}. \tag{4}$$

The availability ratio $a(u)$ is estimated using the join timestamps of the hosts $T_{join} = \{t_{join}(u) \mid u \in G\}$, where $t_{join}(u)$ is the join timestamp of host u, as

$$a(u) \propto \frac{min\ T_{join}}{t_{join}(u)}. \tag{5}$$

A major requirement of the management scheme is that the joint capability value of each host is made available on every other host. This can be done in a static way by reserving a portion of the host identifier, which is otherwise initialized with random data, for capability descriptions (see Figure 6). The major advantage of this technique is its superior performance: there is no need for communication, except for the host identifier itself, which is communicated anyway by the group membership protocol. However, the fact that capabilities are hard coded may force a host to leave and rejoin the group, when a capability value must be updated. Note however, that such additional updates never happen for our joint capability function for two reasons: First, the relative performance won't change without substantial modifications to the hard- and/or software setup of the host, which would probably require a reboot of the machine forcing a rejoin operation. Second, the availability ratio of a host u can change only if either u rejoins or the host v with $t_{join}(v) = min\ T_{join}$ leaves. In either case a can be computed for all hosts without any additional membership updates.

Alternatively, a simple Query/Response-protocol can be used for more dynamic capability functions to explicitly fetch information about the capabilities of a remote host before actually adding the host identifier to the local view. With $O\left(|G|^2\right)$ point-to-point message exchanges for a group G of size $|G|$, this approach suffers from comparatively high communication complexity. Yet, the resulting adaptability can become essential when host capabilities are highly dynamic.

6.2 Competition-based Method

P2P Grids are typically large-scale networks comprised of several hundreds, sometimes even of thousands of nodes. Scalable membership management in such networks is challenging and thus has been extensively researched during the last years. A fundamental finding is that maintaining full membership knowledge for a large number of volatile nodes is not feasible, as even powerful nodes are quickly overwhelmed by permanently fluctuating membership lists. Hence, state-of-the-art membership protocols for large scale networks establish partial local views referencing only a small number of contacts. For topology management within large groups this has immediate consequences: First, schemes based on global knowledge, like the ID-based management scheme described above, are not applicable. Instead, we have to decide which nodes populate upper layers of the aggregation tree based on incomplete information gathered from local or nearby information sources. The second consequence of partial views is that, even if we can allocate higher-level V-Nodes appropriately, V-Nodes do not necessarily know suitable link partners, as they may not be in their (partial) membership list.

To cope with these difficulties our second tree management scheme makes use of auxiliary information sources: First, hosts providing V-Nodes on a certain layer λ join a *layer group* G_λ. Note, that the only requirement on the group model for layer groups is that at least a single other host is reported, if one exists. (However, for performance reasons lightweight group models should be preferred.) By examining the membership list of their layer group,

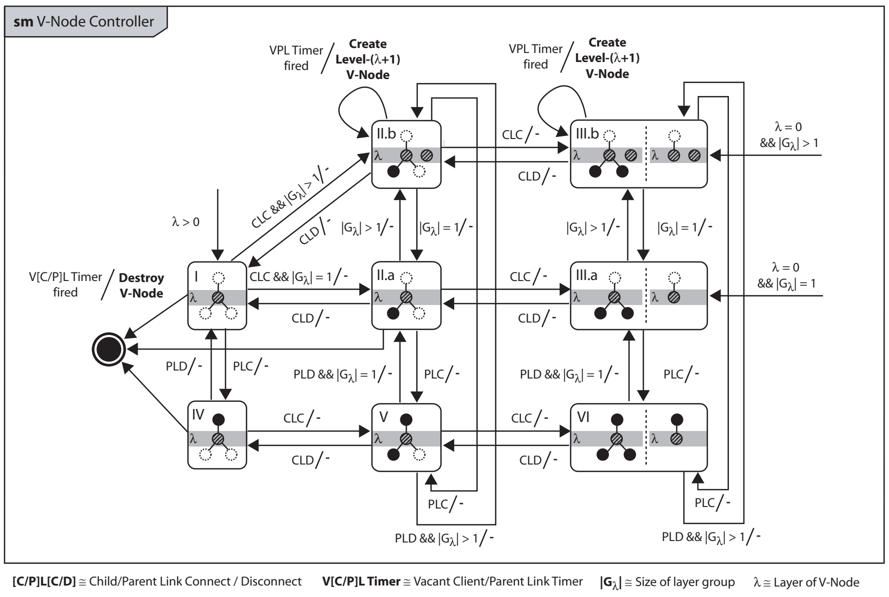


Fig. 7 State machine controlling the lifecycle of V-Nodes (UML Notation). The state icons visualize the connectivity of the owning V-Node: While the upper circle represents the parent link, the two lower ones stand for the child links. Connected links are displayed as discs, vacant links are circles. States II.b and III.b have another state component that is drawn adjacent to the local V-Node and stands for a known group layer neighbor. For example, a V-Node in State II.b has one assigned and one vacant child link, a vacant parent link, and a group layer neighbor is known. V-Nodes in State VI are fully connected, thus both child and the parent links are assigned for V-Nodes on the upper layers and the parent link is assigned for layer-0 V-Nodes.

one can easily detect whether there are other V-Nodes on the same layer. Second, hosts maintain a cache of vacant links. This cache is used to discover potential link partners and is kept up-to-date by an epidemic diffusion protocol. In the following sections, we describe in detail, how these information sources are used to build and maintain the aggregation tree.

6.2.1 V-Node Lifecycle Control

Figure 7 depicts the state machine controlling the lifecycle of a level-λ V-Node. Initially all links of a newly created V-Node are vacant (III.{a,b} for leaf, I for non-leaf V-Nodes). The V-Node controller passes into a new state in the following cases: a link becomes assigned ($[C/P]LC$ trigger), a link gets vacant ($[C/P]LD$ trigger), or the layer group view is updated ($|G|\,[>/=]\,1$ trigger), which means that a V-Node on the same layer has been created or destroyed remotely. States II.b and III.b are special in that the host is *promoted*, i.e., a local level-$(\lambda+1)$ V-Node is created, after a V-Node has been within one of these states for a predefined time $T_{Promotion}$. Promotion happens only, if there is not already a local level-$(\lambda+1)$ V-Node.

As hosts promote independently, there is a risk that too many level-$(\lambda+1)$ V-Nodes are created simultaneously. Thus, we introduce a heuristics that links the *promotion probability* $P\{u \textit{ promotes}\}$ for a level-λ V-Node u to the number of already existing level-λ V-Nodes such that

$$P\{u\ promotes\} \propto \frac{|G_0|}{|G_\lambda| 2^{\lambda+1}} \tag{6}$$

holds for all $\lambda \geq 0$. As this probability decreases with the number of existing level-λ V-Nodes, an oversupply on one layer is less likely transfered to the next higher layer. Note that the actual size of a layer group G_λ is not available for group models with partial membership lists. However, many group models, including SCAMP (see Section 3.2.2), allow for an estimation $|G_\lambda|_E$ of the actual group size that can be used to evaluate Equation 6.

V-Nodes with limited connectivity are removed after a specific period of time: When both child links of a parented V-Node are vacant (IV) or the V-Node is orphaned and has at least one vacant child link (I,II.a) for a time span $T_{Destruction}$, the V-Node is destroyed. As there are less potential link partners for nodes on higher layers, we increase $T_{Destruction}$ with the V-Node layer

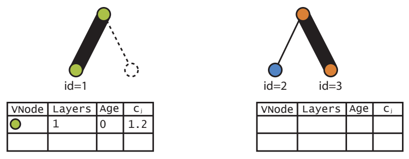
$$T_{Destruction}(u) \propto 1 + k \log \lambda \tag{7}$$

where k is a constant scale factor. Thus, higher level V-Nodes have more time to discover link partners as described in the following section.

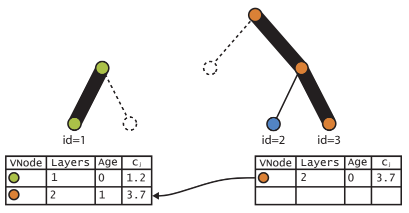
While timeout-based promotion provides for on-demand creation of V-Nodes, destruction in case of continued lack of demand ensures that only those V-Nodes endure that are required to maintain the topology. Both mechanisms together result in V-Nodes on higher levels competing for V-Nodes on lower levels in a market-oriented manner. We thus refer to this scheme as *competition-based*. Note, that the increased scalability of this approach comes at the price, that its efficiency is lower than that of the ID-based tree management scheme.

6.2.2 Link Establishment

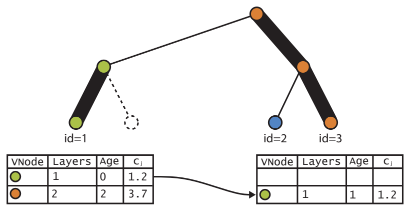
A V-Node tries to connect to V-Nodes on the next higher layer permanently, even if the parent link has been already assigned. If a V-Node with a lexicographically smaller identifier is found, the link is retargeted to that one. Hence, excess V-Nodes eventually become obsolete and can be destroyed. Potential link partners are fetched from a local cache containing descriptions of vacant remote child links. Cache maintenance is done by disseminating unconnected link information within the aggregation group using a gossip-style protocol. Due to their probabilistic nature, gossip protocols can remain very simple while still providing fast convergence, outstanding scalability, and high robustness. For this reason, they have gained popularity in various contexts, including database maintenance [26], probabilistically reliable multicast [27], and aggregation [7]. Every T_{Round} time units, each host pushes descriptors of unassigned local child links and a fraction of cache entries for remote links to another randomly selected host. The receiver merges the arriving entries into the cache by first filling up empty slots and then replacing existing entries. To keep cache information up-to-date, victim entries, which are entries that are replaced when there are no more empty cache slots, are selected based on an *age* attribute. A descriptor starts with an age value initialized to zero when it is inserted by the hosting V-Node. For non-local descriptors, the age counter is increased by one in each round. The increasing probability of being selected as a victim entry, ensures that stale or outdated information is eventually displaced by more current information and vanishes from the caches. Figure 8 illustrates an exemplary link establishment sequence.



(a) Round k: Caches are populated with vacant local child links.



(b) Round $k+1$: V-Node $(3,1)$ was in state III.b. As no level-2 V-Node with vacant child node has been discovered host 3 has been promoted. V-Node $(1,1)$ was in state II.b. Thus host 1 might have been promoted two. However, we assume that a vacant child link of the newly created V-Node $(3,2)$ was discovered first, i.e., the cache of host 1 has been updated with descriptors from host 3.



(c) Round $k+2$: A link between V-Node $(1,1)$ and V-Node $(3,2)$ has been established.

Fig. 8 Link establishment sequence with open child link caches. V-Nodes are identified by aggregation points, i.e., $(host, layer)$ pairs (cf. Section 6).

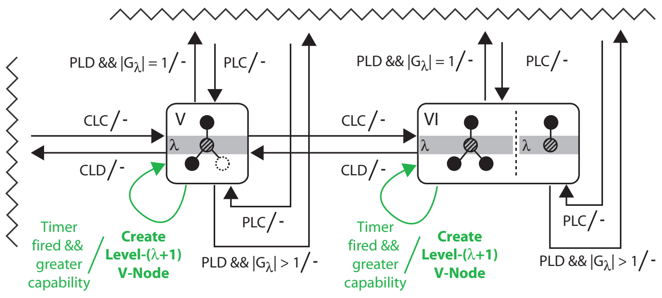

PLD && $|G_\lambda| = 1$ / -
PLC / -
CLC / -
CLD / -
V
λ
Timer fired && greater capability / Create Level-(λ+1) V-Node
PLC / -
PLD && $|G_\lambda| > 1$ / -
PLD && $|G_\lambda| = 1$ / -
PLC / -
CLC / -
CLD / -
VI
λ
λ
Timer fired && greater capability / Create Level-(λ+1) V-Node
PLC / -
PLD && $|G_\lambda| > 1$ / -

Fig. 9 Augmented state machine for controlling the lifecycle of V-Nodes in our capability aware competition-based tree management scheme: Having V-Nodes in state V and VI create a level-$(\lambda+1)$ V-Node when they are more capable than their current parent V-Node ensures that the system can escape from suboptimal steady states.

6.2.3 Capability Awareness

Capability awareness is incorporated into the competition-based management scheme by using the same joint capability function as in Section 6.1.1. Furthermore, we introduce two additional behavioral patterns: First, V-Nodes no longer look for link partners with lexi-

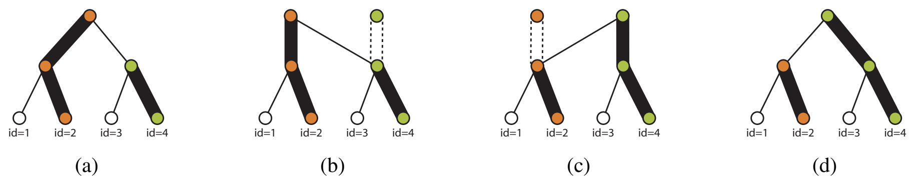


Fig. 10 Transition of a suboptimal 4-node ($c_j(4) > c_j(2) > c_j(3) > c_j(1)$) steady state configuration to the optimal configuration enabled through displacement of less capable V-Nodes and links to them.

cographically smaller identifiers, when their parent link is assigned, but for those that are more capable. Consequently, V-Nodes with higher capability values are more attractive to V-Nodes on the next lower level. To support this behavior, we attach the joint capability value of the hosting node to the link descriptors disseminated for link establishment. A receiving node can then quickly check whether another candidate is more capable than the one it is currently connected to by simple floating point number comparison.

Second, the state machine depicted in Figure 7 is augmented with a new transition for State V and VI (see Figure 9), that creates a level-$(\lambda + 1)$ V-Node when the joint capability value of the local V-Node is greater than that of the newly assigned parent link partner. This modification is necessary to set off suboptimal steady states such as depicted in Figure 10a: Since host 4 is more capable than host 2, the state machine of the level-1 V-Node on host 4 – which is in state VI – creates a level-2 V-Node as soon as the respective timer fires (see Figure 10b). As links are migrated to more capable parents due to our first modification (see Figure 10c) and childless V-Nodes are eventually destroyed, the optimal steady state is finally reached (see Figure 10d). By delaying the actual creation of the V-Node for a certain time, we ensure that it is created only, if no other V-Nodes with equal or higher capability values become available that have not yet been discovered. Together these patterns ensure that V-Nodes hosted on more capable hosts eventually displace those from less capable hosts.

Note, that this scheme supports dynamic capabilities, as capability updates once seeded by a host eventually displace outdated values spread throughout the distributed caches.

7 Sensing Subsystem

We expose a large body of system information out-of-the-box. This includes various OS metrics (see Table 2) collected by third party system information gathering and reporting solutions [28] as well as data obtained from instrumented COHESION components (e.g., the number of open tasks in a distributed task pool [4]). By leveraging the modular design of the COHESION platform, our system supports application-specific extensions. With such an extension mechanism the number of subsystems to monitor and manage can quickly become overwhelming. Hence, our information aggregation solution includes a sophisticated sensor management system described subsequently.

The architecture of the sensor management facility is comprised of three main abstractions that can be used to create sensor/actuator networks of arbitrary complexity: wireables, wires, and a sensor pod (see Figure 11). *Wireables* are the basic building blocks for sensor/actuator networks. They are either *sensors* providing monitoring data or *actuators* translating incoming monitoring data into some kind of action within a target subsystem. For

Domain	Path	Description
CPU	`/system/cpu/info/model`	CPU model description
	`/system/cpu/info/number`	Number of CPUs
	`/system/cpu/info/cache`	CPU cache size in bytes
	`/system/cpu/info/vendor`	CPU vendor description
	`/system/cpu/info/speed`	CPU speed in MHz
	`/system/cpu/time/load_fifteen`	System load average for last fifteen minutes
	`/system/cpu/time/load_five`	System load average for last five minutes
	`/system/cpu/time/load_one`	System load average for last minute
	`/system/cpu/time/wait`	CPU wait time in percent
	`/system/cpu/time/nice`	CPU nice time in percent
	`/system/cpu/time/user`	CPU user time in percent
	`/system/cpu/time/system`	CPU system time in percent
	`/system/cpu/time/idle`	CPU idle time in percent
Memory Subsystem	`/system/mem/free`	Free system memory in bytes
	`/system/mem/total`	Total system memory in bytes
	`/system/mem/used`	Used system memory in bytes
	`/system/mem/ram`	RAM size in MBytes
	`/system/swap/used`	Used swap memory in bytes
	`/system/swap/free`	Free swap memory in bytes
	`/system/swap/total`	Total swap memory in bytes
I/O Subsystem	`/system/disk/stat/total`	Total disk memory in bytes
	`/system/disk/stat/free`	Sum of free disk memory in bytes
	`/system/net/info/hostname`	Fully qualified hostname
	`/system/net/info/ip`	IP address
	`/system/net/stat/bytes_in`	Outgoing bytes per second
	`/system/net/stat/bytes_out`	Incoming bytes per second
	`/system/net/stat/packets_in`	Incoming packets per second
	`/system/net/stat/packets_out`	Outgoing packets per second
Process Management	`/system/proc/running`	Number of running processes
	`/system/proc/sleeping`	Number of sleeping processes
	`/system/proc/stopped`	Number of stopped processes
	`/system/proc/zombie`	Number of zombie processes
	`/system/proc/total`	Total number of processes
Miscellaneous	`/system/os/name`	Name of operating system
	`/system/os/version`	Version of operating system
	`/system/os/uptime`	Machine uptime
	`/system/os/machine`	Machine platform information

Table 2 Our aggregation system provides a comprehensive set of metrics gathered from the operating system.

example, a decrease in available network bandwidth may trigger an actuator to throttle the frequency of work-stealing attempts initiated by the COHESION distributed task pool. Wireables are first class objects uniquely identified by $(path, attributes)$ pairs. As in many other management and monitoring solutions [29,30], the *path* part of the identifier is used to organize the potentially large number of sensors in a tree-structured directory browsable through the sensor pod. The *attributes* part is a set of key/value-pairs that is used to further specify the purpose of the sensor. A CPU usage sensor, for instance, is available twice in a dual-core or dual-processor system. While both share the same *path*, e.g., `/system/cpu/usage`, they can be differentiated by their attributes `cpu.id=1` and `cpu.id=2` respectively.

Sensors and actuators are connected through *wires*, which are used to deliver values measured by the sensor. Delivery may be initiated proactively by the sensor (*push*) or on-demand by the actuator (*pull*). Actuators can attach arbitrary attributes (which are key/value-pairs) to a wire during the establishment process. Since these attributes can subsequently be read by the sensor, this mechanism can be used to pass arguments and hence allows for sensor configuration on a per-wire basis. Note, that this feature is key to prevent measurement duplication (see Section 7.1) and to support sharing of the aggregation infrastructure (see

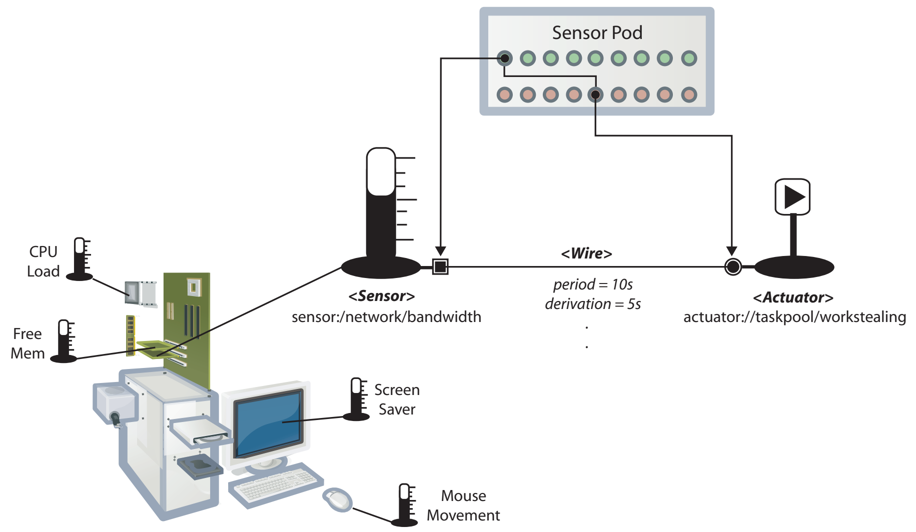


Fig. 11 Components of the sensing framework: *Sensors* gather information from hard- and software components that is pushed to *actuators* over logical *wires*. Wires are created and managed by a *sensor pod* component.

Section 9). The coordination of the wire establishment process is delegated to *wire factories*. Custom wire factories may be contributed by bundles and can be used to extend the framework to support unforeseen use cases (G4).

Since COHESION is a dynamic modular system, the lifecycle of a wireable is intimately connected with the lifecycle of its hosting bundle such that wireables are inherently transient. Thus, hardcoded dependencies are inappropriate when creating possibly large sensor/actuator networks. Instead, nodes must be able to dynamically create, register, lookup, wire up, and finally destroy wireables. However, coping with the resulting dynamism on the application level is complex and error-prone. Thus, COHESION provides a *Sensor Pod* to simplify, coordinate, and automate these management tasks.

Since there may be a large number of wireables, eager wireable instantiation on platform startup would result in an increased startup time and memory footprint violating our requirement to support resource constrained hosts (S2). Thus, wireable instantiation happens on demand: the sensor pod delays creation and activation of wireables until they are actually used, i.e., a wire is going to be attached. *On-demand instantiation* is especially useful when a bundle exposes a large number of sensors that are used only occasionally or when sensor instantiation and operation are expensive, e.g., when external information systems like the *Network Weather Service* (NWS) [31] are integrated. To further simplify sensor/actuator network deployment, *auto wiring* automatically connects sensor/actuator pairs satisfying user-definable conditions. It can be used to implement sensor/actuator networks in a declarative way.

Wireables are implemented as OSGi components registering a `Sensor` or an `Actuator` service with the OSGi service registry. Thus, applications can easily deploy any number of custom wireables to extend the set of information sources available for aggregation (G4). To allow bundle authors to decide what information is exposed to whom (S4), we leverage the ability of the OSGI service registry to restrict access to registered services. Thus, the sensor

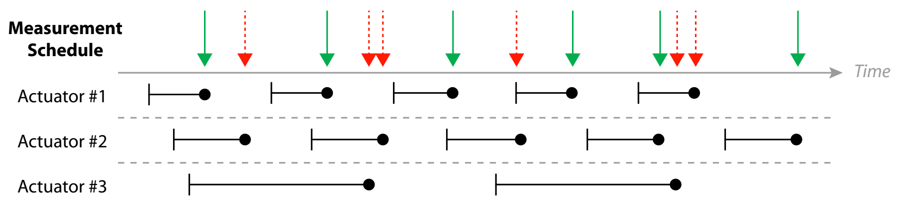


Fig. 12 Prevention of measurement duplication for a sensor with three attached wires. While solid green arrows indicate actual measurements, dotted red arrows represent requested measurements that are serviced by reusing the last value measured. In this example, 50% of the requested measurements can be saved.

pod provides a filtered view on the OSGi service registry, called the *Sensor Bus*. The sensor bus serves as a directory for wireables that can be used by applications to lookup wireables they are interested in.

7.1 Wire Flow Control

Sensor/actuator interaction is stipulated by providing values for measurement rate and deviation tolerance on wire creation. Together they define a time window that is used to detect whether a previously generated sensor value is current enough, i.e., it's timestamp lies within that window (see Figure 12). This simple mechanism prevents duplication of measurements, when more than one actuator is attached to a sensor.

In many situations an actuator is not interested in all values produced by a sensor. For example, a low memory detector is only interested in measurements when the amount of free memory drops below a given threshold. Since wire partners are not necessarily collocated, a mechanism is needed that enables actuators to implement delivery conditions within or collocated to the target sensor. Such conditions are called *filters* in our architecture. When a filter applies, the updated value is not forwarded to the actuator. Filters are evaluated whenever the sensor proactively tries to push a value over the wire. Measurements initiated by a pull operation are delivered irrespective of installed filters. By intercepting a sensor value as early as possible, we can avoid unnecessary transmission of sensor values and evaluations of actuator logic. This advantage is particularly pronounced when sensor and actuator are not collocated (see Section 7.2).

Filters are plain Java objects implementing the `Filter` interface, which requires a single method `void accept(T value)` to be implemented. Support for declarative filter definition is available through filter implementations where conditions are expressed through LDAP or script language expressions. Thus, filter behavior can be adjusted without recompilation of the hosting bundle.

Besides *filter by value*, a second use case of filters is to suppress delivery of measurements where the value has changed in a way that is not significant or where the value has not changed at all. The low memory detector, for instance, is not interested in minimal changes of free memory when the new value differs only by a couple of bytes. Such use cases are prime examples for our system's ability to *filter by change*. The respective wire filter accepts only values that differ from the last value by an absolute amount (e.g., the amount of free memory has changed by 1 MB) or a relative amount (e.g., the amount of free memory has changed by 10%).

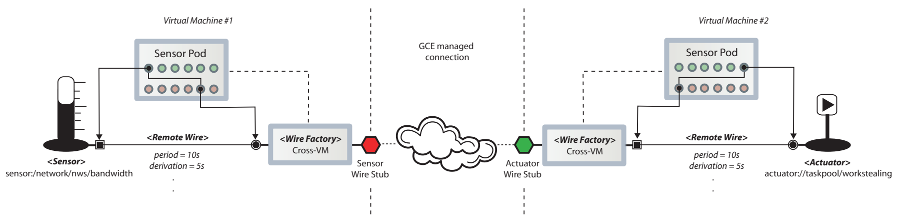


Fig. 13 Remote wires are realized by deploying transparent wire stubs that establish a connection within the virtual overlay network using COHESION's *Generic Connection Establishment* (GCE) facility.

Wire flow control ensures host system resources are used most efficiently (G2), while still allowing for maximum flexibility (G4).

7.2 Cross-VM Wires

Although we use sensor/actuator networks for many purposes, including application lifecycle control [4] and link latency detection for victim selection in work stealing algorithms, a key use-case is the delivery of data within the reducer network used for aggregation. For that purpose we need a mechanism to establish cross-VM wires, which are wires that connect a sensor deployed in one Java Virtual Machine (VM) with an actuator in another VM (see Figure 13).

Technically, our solution is implemented by a specialized wire factory providing stubs on both ends of the wire. While the stub collocated with the actuator (called the *actuator wire stub* (AWS)) is created locally to the requesting client, the wire factory implementation creates a *sensor wire stub* (SWS) in the VM of the target sensor. After the stubs have been created, the AWS establishes a connection to the SWS within the COHESION overlay network using a generic connection establishment (GCE) facility. Thus, aggregation trees can span arbitrarily segmented physical networks (S4). GCE hides the details of the connection establishment and maintenance process (handshake, half-open connection detection and connection shutdown). As wire attributes and filters are transmitted as part of the handshake process, the fact that the wireables are connected to a non-local partner is completely transparent. Hence, we can use the wire flow control mechanisms described above to control the aggregation process in a fine-grained way.

7.3 Measurement Scheduler

Every phase in the aggregation process (cf. Section 5) consumes resources. In a P2P Grid scenario this can cause problems: First, a P2P Grid application may run in parallel to other processes on the host system sharing a common limited pool of resources. Hence, the resource consumption of the P2P Grid application must be limited to avoid that the user experience is impaired (S2). Second, hosts are highly heterogeneous in a P2P Grid (S3). Measurement schemes tailored for one host may easily overstress less powerful hosts. To circumvent these problematic situations, our sensor management facility provides a measurement scheduler that allows to limit resource consumption by specifying quotas for each type of resource and per-sensor priorities to control which sensor consumes which part of the

Fig. 14 Resource share assignment algorithm based on lottery scheduling.

```
1: procedure CREDITSHARE(rid)
2:     $sensors \leftarrow$ GETSENSORS($sensorPod$)
3:     $tickets_{overall} \leftarrow 0$
4:     for $i \leftarrow 1$, SIZE($sensors$) do
5:         $sid \leftarrow$ GETID(SENORS[$i$])
6:         $tickets_{overall} \leftarrow tickets_{overall}$ + GETPRIORITY($sid, rid$)
7:     end for
8:     $ticket \leftarrow$ RANDOM($0 \ldots tickets_{overall}$)
9:     $c \leftarrow 0$
10:    for $i \leftarrow 1$, SIZE($sensors$) do
11:        $sid \leftarrow$ GETID(SENORS[$i$])
12:        $c \leftarrow c$ + GETPRIORITY($sid$)
13:        if $c >= ticket$ then
14:            $shares[sid, rid] \leftarrow$ GETQUOTA($rid$) $* \frac{1}{T_{assign}}$
15:            break
16:        end if
17:    end for
18: end procedure
```

available resource share. When a sensor can't acquire enough resource shares to perform a scheduled measurement, the measurement is skipped preserving the shares already acquired for the next attempt. Note, that skipping a scheduled measurement does not slow down or stall the overall aggregation process as the reducer network operates asynchronously (see Section 9). However, as no new sensor value has been created, the old value is reused by the reducer on the next higher level. Although we have only implemented CPU time and network bandwidth resource types for the purpose of this paper, our scheduler allows to plug-in arbitrary additional resource types. Each resource implementation must provide a method that is used to predict how many resource shares are necessary for the next task execution. While this mechanism obviously can't enforce hard resource consumption limits, it still ensures compliance when resource utilization is averaged over several executions.

Our scheduler implementation is based on lottery scheduling [32]. This randomized scheduling mechanism allows control over relative execution rates and intrinsically prevents starvation. The simplified routine shown in Figure 14 is executed every T_{assign} time units (the default value is 10 ms) for each resource. First, the algorithm fetches all deployed sensors using the sensor pod (line 1) and sums up their priorities resulting in the overall number of tickets $tickets_{overall}$. After the winner ticket has been drawn (line 8), the associated sensor is computed by recomputing the priority sum, stopping as soon as the subtotal exceeds the ticket number. Finally, the resource share, i.e., the product of $T_{assign}$and the assigned resource quota, is attributed to the winning sensor. Figure 15 illustrates the process for two sensors competing for CPU time. The algorithm actually used in our solution is optimized for speed: First, it lazily recomputes $tickets_{overall}$ only when a sensor is de-/registered or a priority is updated. Second, the winner calculation loop is executed only once, operating on an array of winner tickets, i.e., one winner per resource type.

8 Virtual Tree Topology Management Architecture

In this section, we describe the design and implementation of our *Virtual Tree Topology Management* (VTTM) framework. Both tree management methods presented in Section 6 require that the framework supports a number of basic management operations on V-Nodes.

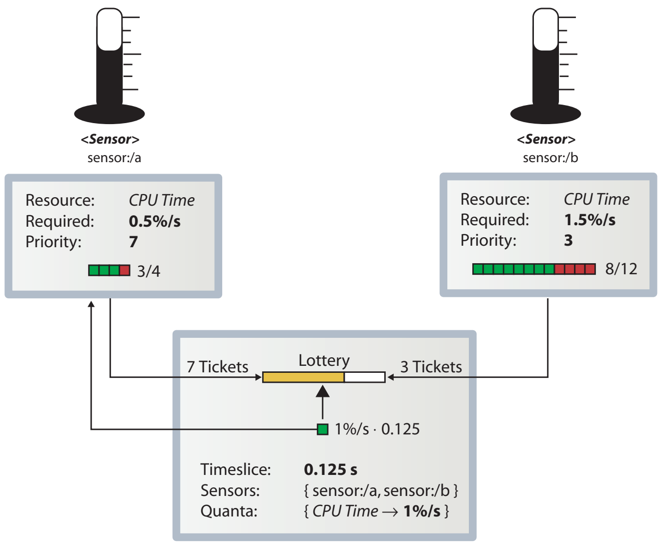


Fig. 15 The lottery based resource assignment process for two sensors. `sensor:/a` wins the CPU time share lottery. As the awarded share satisfies its requirements, the next measurement request will be processed.

As defined in Section 6, V-Nodes are inner nodes of an overlay tree structure which are simulated by hosts. They are uniquely identified with respect to the hosting COHESION node, which is in turn uniquely identified within the group it belongs to.

The management operations on V-Nodes include creation and destruction of trees and V-Nodes at runtime and link establishment between V-Nodes on different levels. A concrete tree management scheme employs a strategy to control when and where to create/destroy V-Nodes and how they should be interconnected. Amongst others, these strategies can be implemented either explicitly in Java (ID-based method) or as a state machine (competition-based scheme). Anyway, VTTM must provide abstractions to model the topology on which these strategies can operate on. As these abstractions must be constructed/destroyed at run-time, stored in variables, and passed as arguments, they are realized as first-class objects, which means they are represented at the language-level as ordinary Java objects. Besides V-Nodes, the remaining VTTM abstractions are *V-Tree* and *V-Links* defined as follows:

V-Tree. A *virtual tree* or *V-Tree* is a container for *V-Nodes* that can be dynamically created and destroyed. V-Trees are scoped by a hosting group to foster isolation and are uniquely identified by the tuple $(group, name)$. An application or component developer is free to instantiate as many topologies as required.

V-Link. Each V-Node can define a set of *virtual links* or *V-Links*. A V-Link is defined by its *role* for the owning V-Node (i.e., CHILD and PARENT). They can be either VACANT or ASSIGNED. In the latter case a (remote) partner is assigned to the local link and vice versa. Note that V-Links are pure logical links, i.e., no physical link is established.

Instances of these abstractions are created through the owning entity. The V-Tree acts as a factory for contained V-Nodes. The same relation holds for V-Nodes and V-Links.

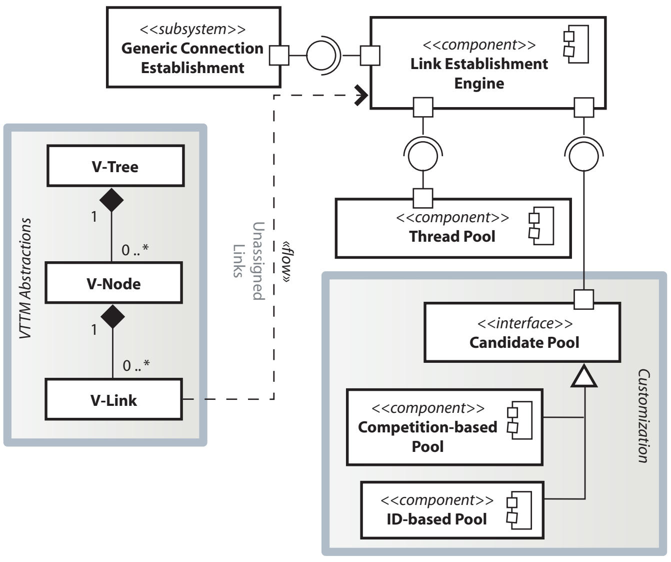


Fig. 16 Component diagram for VTTM link establishment infrastructure.

8.1 Link Establishment

While creating and destroying V-Nodes is straightforward and is thus directly controlled by the tree management scheme, assigning of V-Links is a potentially complex task, since the process must be coordinated between the prospective link partners. Thus, VTTM provides a flexible infrastructure (see Figure 16) that supports a wide range of link partner selection strategies. A *Link Establishment Engine* (LEE) monitors the state of all V-Links for a given tree. For each vacant link the identifier of a potential link partner is fetched from a *Candidate Pool*. As the identifier contains the host ID, the engine can initiate a connection attempt. On success, the V-Link is assigned. Many V-Trees will have more than a single unassigned V-Link. To speed up tree construction, VTTM allows for fully concurrent link establishment: engines can handle multiple parallel incoming connection attempts and are trying to establish links with multiple other V-Nodes concurrently by employing a pool of worker threads. If any attempt is successful, the link becomes assigned and all other connection attempts for the same link are immediately aborted. Hosts can tune the level of concurrency and thus can trade off link construction speed against resource consumption. To implement the schemes presented in Section 6, we provide custom candidate pools: While the pool for ID-based tree management can compute the single uniquely identified partner for any link by simply evaluating the total order R imposed on the set of available nodes, the one for our competition-based tree management scheme uses information available from the local vacant child link cache.

8.2 Resolver Service

VTTM provides a generic query mechanism that can be used to collect information from V-Nodes by propagating queries along the edges of the tree (see Figure 17). Query evaluation

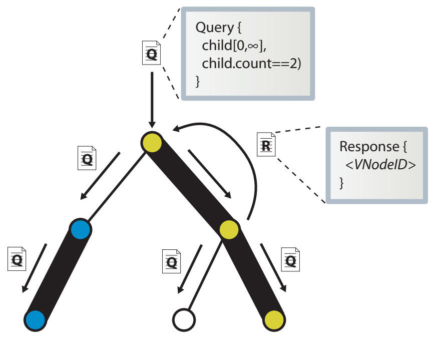


Fig. 17 Processing of a simple VTTM query that is forwarded along all child links. It returns the local V-Node identifier, if the V-Node has two assigned child links.

on each V-Node is a double-stage process: First, it is evaluated, whether the query should be actually executed locally, forwarded along the currently selected V-Link, or both. If the first phase yields that the query should be executed, this happens in the second phase and the result is sent to the initial source of the query, where a callback is invoked to handle the incoming result. We use the resolver service to support MAPS, which are introduced in the next section.

9 Hierarchical Aggregation Service

In our system, every node can request any number of aggregations by issuing an *aggregation request* through an *aggregation service*. An aggregation request consists of a sensor identifier and a set of aggregation points (see Section 6), e.g., $(sensor: /network/bandwidth?interface = eth0, \{(xmpp : //f28aa1@cohesion.de : 80/f28aa1 - 35, 2)\})$. While the former specifies the sensors whose output should be aggregated, the latter specifies the set of V-Nodes from which aggregated values are to be fetched. Due to the dynamism prevalent in P2P Grids, the selection of V-Nodes from which to fetch aggregated values is no straightforward task. For example, a host may be interested in receiving aggregate values from all V-Nodes at a certain level within the aggregation tree. Since nodes come and go unpredictably, the composition of the set of matching V-Nodes is in a constant state of flux. Exposing that dynamism at the application level would considerably increase application complexity. Thus, we provide *Managed Aggregation Point Sets* (MAPS). A MAPS is a dynamic set of V-Nodes, which automatically adapts to changes in the underlying tree topology. MAPS' make use of the topology resolver service to find V-Nodes satisfying a set of custom constraints defined by the application. For example, we use a MAPS to get all V-Nodes on a given level in our monitoring application (see Section 11).

9.1 Reduction Tree

The aggregation system deploys a *reduction tree* to serve aggregation requests. A *reducer* is located at every V-Node of the aggregation tree. A *level-λ reducer* is a component composed of an actuator, that consumes data from level-$(\lambda - 1)$ reducers, applies the reduction function to the set of collected values, and delivers the result to level-$(\lambda + 1)$ reducers. To

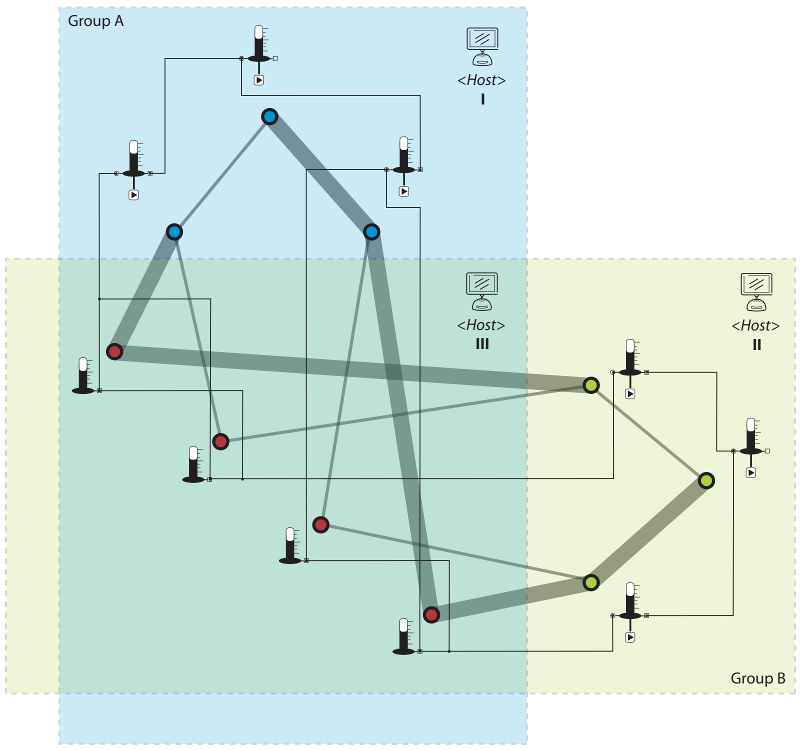


Fig. 18 Inter-Group isolation and intra-group sensor-based sharing of the reduction network.

support on-demand instantiation, reducers are implemented as a sensor factory. This means that a reducer is created as soon as someone tries to create a wire to it (see Section 7). In addition, wires attached to the sensor of a reducer are automatically replicated on the next aggregation layer. For each incoming wire of the sensor of a level-(λ) reducer, an outgoing wire with the same wire attributes (including filters) is created and connected to the level-($\lambda - 1$) reducer. Hence, the creation of a wire to a set of top-level reducers defined by some MAPS object, results in the creation of a reduction network, consisting of all reduction trees mounted at the top-level reducers of the aggregation request. As wire attributes are propagated down the aggregation tree, the target sensor is configured to push values periodically at the rate specified when creating the wire to the top-level reducer. Wire propagation decouples reducers and thus allows to respect resource consumption limits enforced by the measurement scheduler (see Section 7.3). Since reducers are disposed as soon as no incoming wire is attached for a given period of time, wires are progressively destroyed down the aggregation tree. Hence, the whole reduction network is eventually garbage-collected when all wires are detached from the top-level reducers.

9.2 Isolation and Sharing

Maintaining the aggregation infrastructure consumes resources. Thus, in order to use the infrastructure most efficiently (G2), it should be shared whenever possible without eroding isolation. Since in COHESION services are provided on a per-group basis, aggregations requested by different applications are isolated, preventing any undesirable interference across applications (Host I and Host II in Figure 18). However, the aggregation infrastructure can still be shared among different applications simply by using a shared group (Hosts I/II and Host III in Figure 18). As sensor visibility may be restricted to certain groups, inter-group isolation enforces access control as required by (S4).

Our system establishes a single aggregation tree per group. For example, requests issued by Host I and Host III are processed using the same shared aggregation tree located in Group A. Hence, the number of comparatively expensive tree maintenance operations is reduced to a minimum. On top of that, a reduction tree is deployed for each sensor, i.e., reducers are used to process all aggregation requests for that sensor. Note that reducers are simple wireables. Hence, reducer networks are lightweight, and maintaining a private network for each sensor causes not much stress on the system.

9.3 Storage and Propagation of Measurements

In many aggregation use cases a history of aggregated values is required. The required storage for time series' of measurement data is called a *Management Information Base* (MIB) [7,33]. We use a round-robin database [34] as MIB, so that system storage footprint remains constant over time. To avoid the overhead of MIB maintenance, we do not update MIBs by default. Instead, an on-demand instantiated actuator that is auto-wired to reducers can be deployed, when aggregation history is actually required.

Often other nodes than the requester are interested in receiving aggregate values. Thus, an aggregation system can be extended to provide a service for propagating aggregate values. SDIMS [8] provides a very flexible implementation of such a service as part of their *Update-Up*`k`*-Down*`j` aggregation and propagation strategy. For a pair (k,j) aggregation is performed up to the k-th level and aggregate values are propagated downward for j levels. Our approach is even more flexible, as aggregation source V-Nodes are specified by a MAPS object, that can reference any subset of the V-Nodes of the aggregation tree. Propagation can be done either by groupcasting aggregate values or by using the resolver service to deliver values along the edges of the aggregation tree. Note that the former is in general much more efficient in P2P Grids when all nodes are interested in updates, since efficient low-level broadcast techniques (e.g., broadcast domains in LANs) can be used (at least within network segments). Since the propagation strategy in our system is not tightly coupled with the aggregation strategy as in SDIMS, we can even support things like conditional propagation, where aggregate values are propagated only if a certain condition holds. As in the case of MIB maintenance, propagation is implemented as an on-demand instantiated actuator that is auto-wired to reducers.

10 Evaluation

To substantiate our claims concerning efficiency and scalability of our aggregation system and to demonstrate the benefit of integrating capability-awareness into tree management schemes, we conducted comparative studies on a compute cluster consisting of Intel Xeon 2.66 GHz processors. Our evaluation supports three main conclusions: First, our results confirm the assumption that the ID-based tree management scheme would clearly outperform the competition-based scheme, because it was primarily designed for scalability. Second,

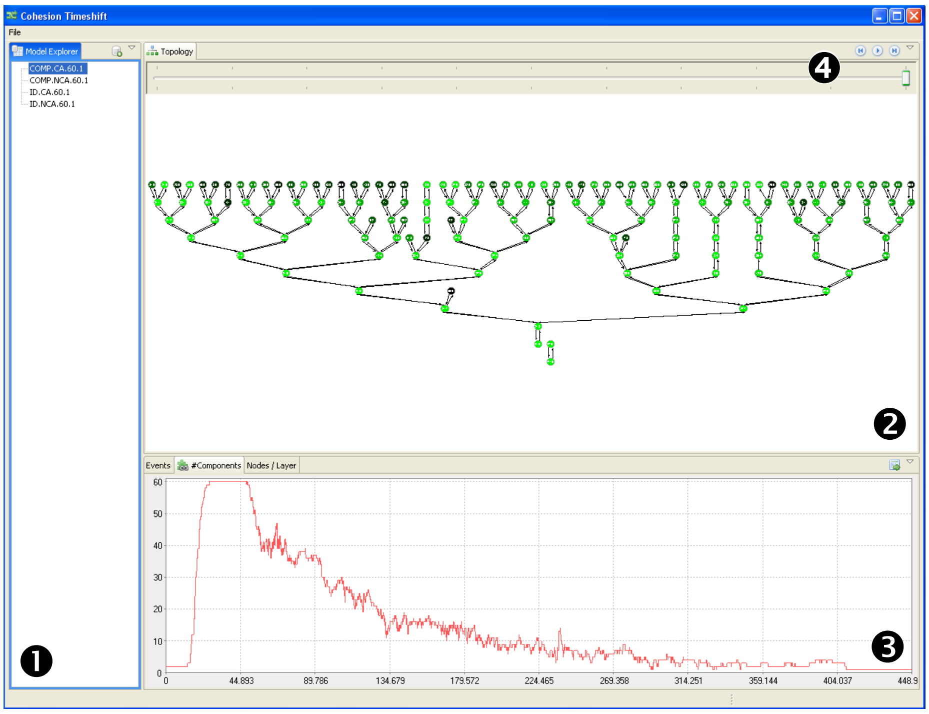


Fig. 19 *Timeshift* is an extensible tool for post mortem analysis of multi-sourced event streams. It allows for analysis (❸) and comparison of multiple event streams (❶) and the resulting topologies (❷) over time (❹).

capability-awareness moderately increases construction time, but generates trees of significantly improved quality compared to their capability-agnostic counterparts. Third, our aggregation system scales as expected: CPU utilization per host grows logarithmically with respect to the group size, and linearly with respect to the aggregation frequency and the number of attributes that are aggregated concurrently.

10.1 Tree Construction

Before we discuss our results concerning tree construction in detail, we elaborate on the evaluation methodology and analysis tooling, which we believe, might be of interest to other researchers in the field of tree or more generally topology management.

10.1.1 Methodology

Understanding the distributed process of tree construction is a challenging task that can be substantially simplified by applying visualization techniques. As online visualization techniques are limited to setups with a small number of nodes, we use a post mortem analysis approach: Each participating host writes events, i.e., V-Node creation and destruction, V-Link creation, assignment, unassignment and destruction, to a local event database. The databases of all participating hosts are collected after the test run has finished and are fed into our tool called *Timeshift* for offline analysis (available for download including examples from our website [35]). Figure 19 shows a screenshot of our tool. The tool generates a single event stream by merging events from all host databases. To allow for easy comparison of different executions, *Timeshift* allows to examine several event streams simultaneously. The currently active stream is chosen from a list of available streams (❶). We use the popular graph library *yFiles* [36] to provide a visualization of the aggregation tree (❷). The time slider at the top of the topology view (❹) is used to navigate within the event stream. On change, the underlying graph is modified by processing all events that happened between the old and the new position. The tool supports smoothly animated real-time playback of the event stream. This feature excellently supports the process of building a mental model of the tree establishment algorithms and enabled us to reveal inefficiencies in our algorithm design and bugs in their implementation. Finally, we have incorporated support for pluggable analyzer modules that are used to automatically distill metrics from the event stream. We have implemented analyzers for the following metrics:

(NC) Number of Components. Computes the number of *leaf-containing components* $nc(G,t)$ in the graph G over time (❸). A leaf-containing component is a component that contains at least one leaf, i.e., level-0 V-Node. A fully established aggregation tree has a single component spanning all leafs.

(AQ) Allocation Quality. Computes an indicator q_{alloc} that describes how good the allocation of V-Nodes in a given tree is. q_{alloc} is defined as

$$q_{alloc}(G) = \frac{\sum_{v \in V(G)} \lambda(v)\, c_j(host(v))}{\sum_{v \in V(G)} \lambda(v)}, \tag{8}$$

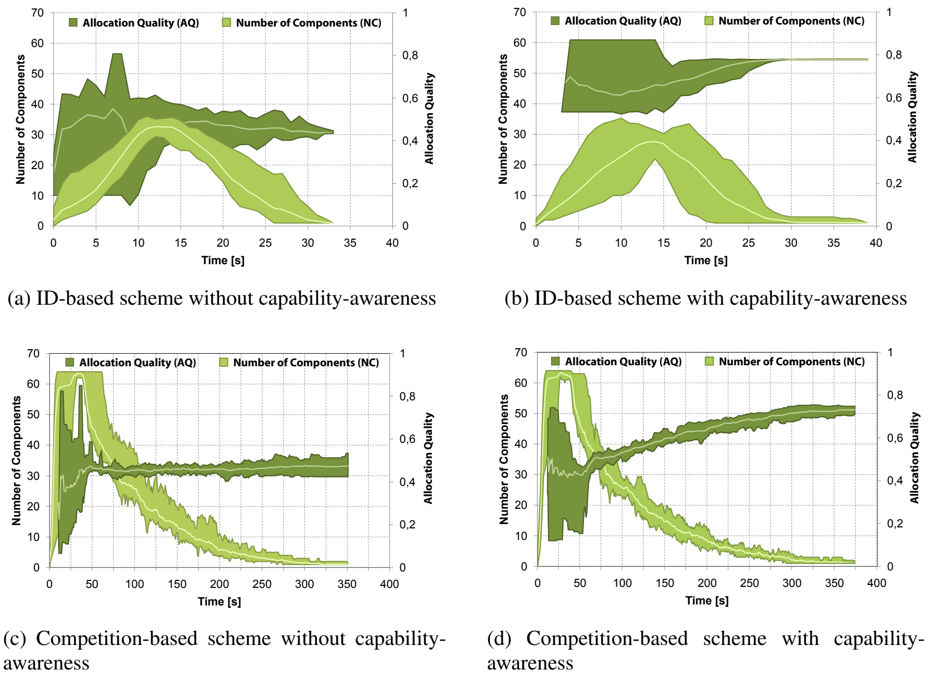
where $V(G)$ is the set of all V-Nodes in G, $\lambda(v)$ is the level of V-Node v, $host(v)$ is the host which simulates v, and $c_j(u)$ is the joint capability value of a host u as defined in Section 6. Note, that this definition of q_{alloc} penalizes allocations of incapable hosts on higher levels.

10.1.2 Results

To assess the performance and quality of our tree management schemes, we have chosen a scenario with 64 hosts building the aggregation tree completely from scratch. Figure 20 shows the NC and AQ metrics described above for both the ID-based and the competition-based tree management schemes with and without capability-awareness over time. For the purpose of simulating a heterogeneous test environment, the joint capability value $c_j(u)$ is computed by mapping the SHA digest of the hostname into the interval $[0,1]$. Thus, we have uniformly distributed values that persist across executions.

Completion Time We first look at completion time, which is the time that elapses until there is only a single component left or, in other words, $nc(G,t)$ drops to one (this includes the time necessary for creating and joining the aggregation group). Despite the overhead for transmitting the joint capability value as part of the host identifiers, there is no significant slowdown for the ID-based tree management scheme with capability-awareness. The slightly higher variation of $[21s, 38s]$ with capability-awareness compared to $[26s, 34s]$ without capability-awareness is negligibly small in relation to the average execution times of $27s$ with capability-awareness and $29s$ without capability-awareness, respectively.

For the competition-based scheme, we observe an increment of approximately 10% that is due to the fact that more capable hosts eventually displace less capable ones, which causes



(a) ID-based scheme without capability-awareness

(b) ID-based scheme with capability-awareness

(c) Competition-based scheme without capability-awareness

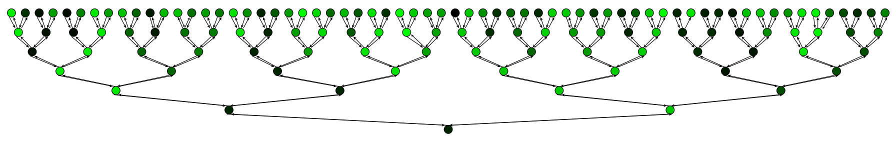
(d) Competition-based scheme with capability-awareness

Fig. 20 Number of leaf-containing components (NC) and allocation quality (AQ) for our tree management schemes. The graphs show the mean value and the range between the minimum and the maximum values within the data set over time.

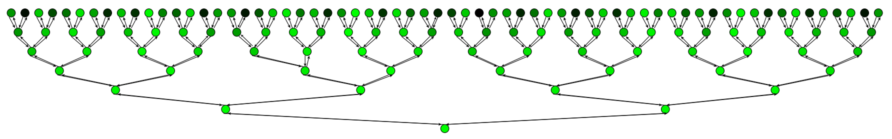
many small tree reconfigurations slowing down the overall construction process. Completion times range within $[275s, 351s]$ without capability-awareness compared to $[294s, 379s]$ with capability-awareness and average to $300s$ and $331s$, respectively.

The fact that the competition-based scheme is outperformed by the ID-based scheme by a factor of ten is due to the following reasons: First, the ID-based scheme is optimal as creating the topology is just a matter of joining the aggregation group. Since in any topology management scheme each host at least has to join the aggregation group, the completion time of the ID-based scheme actually is a lower bound. Second, the gossiping approach for vacant link announcement involves $O(\log(|G|))$ dissemination time [37]. As the number of potential link partners decreases with the V-Node level, the time to find a suitable partner grows towards the root. This observation is substantiated by the fact that half of the links have been established after $\approx 75s$ and the tree construction process decelerates considerably afterwards (see Figure 20c). Third, promotion of a host across the tree levels does not happen interleaved but incrementally. Fourth, we have linked the promotion probability of a level-λ V-Node to the number of already existing level-λ V-Nodes (see Equation 6 in Section 6.2.1) to inhibit oscillations. However, this comes at the price that $P\{u\ promotes\}$ is less than 1 and decreases when the number of level-λ V-Nodes approaches 2^{λ}. Hence, the time that elapses until a V-Node ready for promotion actually promotes can be a multiple of $T_{promote}$.

Allocation Quality If we look at the allocation quality, we observe significantly better values for the capability-aware schemes, namely 0.8 for the ID-based and 0.75 for the competition-



(a) ID-based scheme without capability-awareness.



(b) ID-based scheme with capability-awareness.

Fig. 21 Final aggregation trees for a 64 hosts setup. Darker V-Nodes are located on less, brighter V-Nodes are located on more capable nodes. Capability-awareness obviously results in a significantly improved allocation quality (more brighter V-Nodes towards the root V-Node in (b) than in (a)).

based scheme. Note, that the allocation quality is by construction perfect for the ID-based scheme. In contrast, the average allocation quality of the capability-agnostic schemes are 0.45 for the ID-based and 0.5 for the competition-based scheme.

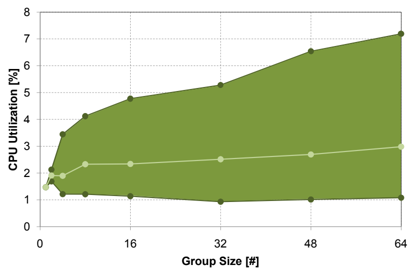
Figure 21 shows an exemplary V-Node allocation for both the ID-based scheme with and without capability-awareness for a fully constructed binary aggregation tree in a 64 host setup. Allocations for the competition-based scheme are similar and are omitted for the sake of brevity here. From these illustrations it becomes obvious that the capability-aware schemes are much more robust in the face of host departures, as higher levels of the tree are simply less likely affected.

10.2 Aggregation

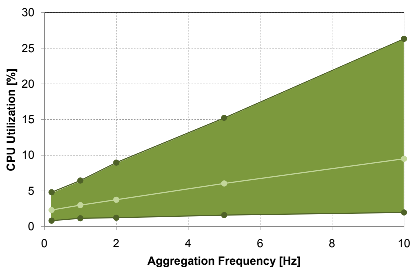
To verify the scalability of our system, we conducted a series of tests evaluating its behavior when the number of nodes within the aggregation group, the frequency with which aggregates are computed, and the number of distinct aggregations performed concurrently are scaled. Figure 22 shows the results of this evaluation.

Group Size Figure 22a shows the CPU utilization for differently sized aggregation groups and an aggregation frequency of 1Hz. While the minimum CPU utilization is independent of the group size and remains roughly constant at 1%, the logarithmic growth for the average and the maximum CPU utilization is evident. This result is no surprise as CPU utilization per V-Node is constant and the most utilized host creates $O(\log_2 |G|)$ V-Nodes. These numbers indicate that our system can be expected to scale up to thousands of hosts before CPU power will become an issue.

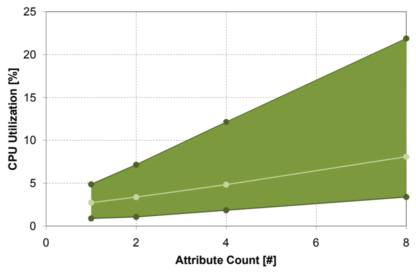
Aggregation Frequency Figure 22b shows the CPU utilization for different aggregation frequencies within a 64 host aggregation group. As before we have traced CPU utilization for the least utilized, the most utilized, and the average host. As expected, CPU utilization grows linearly with respect to the aggregation frequency for all three host types. According to these



(a) CPU utilization for aggregation groups of increasing size



(b) CPU utilization for increasing aggregation frequency



(c) CPU utilization for increasing number of concurrent aggregations

Fig. 22 CPU utilization during aggregation. The graphs show values for the least utilized host, the most utilized host and the average utilization over all participating hosts.

results, our system supports (abusive) aggregation frequencies beyond 10Hz. CPU stress for more realistic use cases with frequencies between 0.1Hz and 1Hz ranges between 2.3% and 3% for the average host and between 4.8% and 6.5% for the most utilized host.

Attribute Count The last scalability characteristic covered by our evaluation is the number of attributes that are aggregated concurrently. Figure 22c shows that CPU usage for different attribute counts within a 64 host aggregation group and an aggregation frequency of 0.1Hz. The numbers clearly show that CPU utilization for the least utilized, most utilized, and the average host grows linearly with the number of attributes. Extrapolating from these results, we can expect our system to support a sufficiently large number of concurrently aggregated attributes.

11 Use Case: System Monitoring

To demonstrate the feasibility of our solution, we have implemented a system monitoring application based on the information aggregation system described above. It consists of two parts: a COHESION bundle that exposes functions of the aggregation service, called the *Monitoring Gateway*, and an Eclipse plugin that connects to this bundle, handles node and metric selection and visualizes the collected data (see Figure 23).

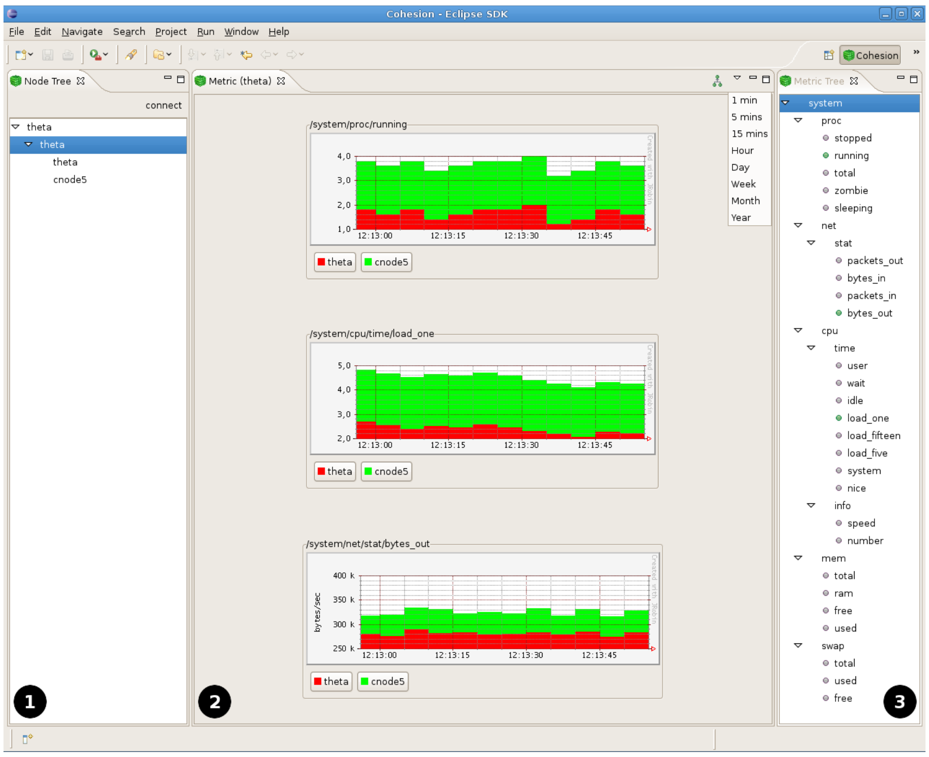


Fig. 23 Our monitoring application based on Eclipse with tree navigation (❶), visualization of several attributes (with history) for the two subtrees mounted at the currently selected V-Node (❷), and a list of all available sensors (❸).

11.1 Monitoring Gateway

The Monitoring Gateway makes it possible to use certain aspects of the aggregation service remotely. Functionality of the gateway includes: requesting the root V-Node of the aggregation tree, getting the children of a given V-Node, and receiving a list of available metrics (sensors). Aggregation on a given V-Node for a given sensor and time period can be started or stopped, and stored aggregation data can be requested.

To be used as an entry point for browsing the system, the root V-Node of the aggregation tree is determined by using the resolver service (see Section 8.2) by simply walking along the edges of the tree until no further parent node can be found. Children for a V-Node can be queried directly using a given aggregation point. Together, these mechanisms are necessary to allow navigation inside the aggregation network. Finally, a list of available metrics is obtained by listing all locally available sensors using the sensor pod (see Section 7) assuming that they are the same on all nodes.

Requests for metrics are forwarded to the aggregation service triggering the creation of a new reducer network for the respective sensor. Depending on the selected V-Node being a leaf in the aggregation tree or not, the aggregation request contains either a single aggregation point or a dynamically updated MAPS (see Section 9), comprising all of its children. By default, metrics are summarized by addition at each inner V-Node. To allow for later modification or cancellation, requests are stored using a unique request identifier.

11.2 Eclipse Monitoring Plugin

The Eclipse Monitoring Plugin connects to the Monitoring Gateway using JMX [38]. The same RRD implementation used as the MIB (cf. Section 9.3) is used to store and prepare aggregated metrics for visualization. The graphical user interface is implemented as an Eclipse *perspective* consisting of three views (see Figure 23):

Node Tree View (❶). Shows the active V-Node, its children (if any) and the path to the root V-node. It can be used to navigate the aggregation network, thereby moving all running aggregations to the currently selected V-Node. On startup, the active V-Node is set to the root V-Node. Changes in the aggregation tree structure are automatically reflected.
Metrics View (❷). Displays graphs of currently active aggregations. Each graph contains aggregated values from all children of the V-Node currently selected in the node tree view. These values are stacked on each other and drawn in different colors. Graphs are updated as soon as new values become available. The amount of displayed metrics history can be chosen from a drop down menu and ranges from one minute to one year. Each child V-Node can be selected to become the new active one by pressing the respective button underneath a graph.
Metric Tree View (❸). Lists available metrics and toggles their activation status. For easier navigation, metrics are arranged in a hierarchical tree determined by the unique identifier of the corresponding sensor (cf. Section 7). On activation, a metric is marked with a green light and becomes available inside the metrics view.

12 Conclusion

In this paper, we present the design, architecture, and implementation of an information aggregation system for P2P Grids. In contrast to general information systems, P2P Grids are used to solve computational hard problems in mid-scale setups and thus often require most efficient instead of most scalable algorithms. To satisfy both requirements, we propose two tree management schemes: one focusing on efficiency, the other on scalability. As resources in P2P Grids are heterogeneous in terms of capability (processing speed, network bandwidth, etc.) and volatile, we designed our tree management schemes to respect these characteristics, resulting in an increased overall performance, especially in the face of tree reconfigurations on node arrivals and departures. To support a broad range of applications, our architecture was designed to be exceptionally flexible concerning all phases of the aggregation process. To limit resource consumption (e.g., CPU time, network bandwidth, and secondary storage capacity) by the information aggregation subsystem, we employ a measurement scheduler based on resource quotas and lottery scheduling. Thus, our approach respects the resource owners sovereignty over his system.

Acknowledgments

Sven Schulz and Hannes Hannak are supported by Deutsche Forschungsgemeinschaft (DFG) under grant BL 941/1-1 and BL 941/1-2.

References

1. Internet Systems Consortium. Internet domain survey - host count, January 2008. *http://www.isc.org/index.pl?/ops/ds/.*
2. Robbert van Renesse. The importance of aggregation. In *Future Directions in Distributed Computing*, volume Lecture Notes in Computer Science 2584, pages 87–92. Springer-Verlag, Heidelberg, April 2003.
3. John Risson and Tim Moorsa. Survey of research towards robust peer-to-peer networks: Search methods. *Computer Networks*, 50(17):3485–3521, 2006.
4. Sven Schulz, Wolfgang Blochinger, Markus Held, and Clemens Dangelmayr. COHESION - A microkernel based desktop grid platform for irregular task-parallel applications. *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Applications*, 24(5):354–370, 2008.
5. Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 19–28, 2004.
6. Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.
7. Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.
8. Praveen Yalagandula and Michael Dahlin. A scalable distributed information management system. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 379–390, 2004.
9. Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 131–146, New York, NY, USA, 2002. ACM Press.
10. Derrick Kondo, Michela Taufer, Charles L. Brooks, Henri Casanova, and Andrew A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 26, Sante Fe, New Mexico, 2004.
11. David P. Anderson. BOINC: a system for public-resource computing and storage. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 365–372, Pittsburgh, USA, November 2004.
12. Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel Distributed Computing*, 63:597–610, 2003.
13. Yoshio Tanaka Kazuyuki Shudo and Satoshi 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)*, pages 259–265, Cardiff, UK, 2005.
14. Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya 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)*, pages 1–12, London, UK, 2002. Springer-Verlag.
15. David P. Anderson and Gilles 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)*, pages 73–80, Singapore, 2006.
16. Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS03*, pages 118–128, 2003.
17. Wolfgang Blochinger, Clemens Dangelmayr, and Sven 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)*, pages 49–56, Singapore, May 2006.
18. Sven Schulz and Wolfgang 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, May 2007.
19. OSGi Alliance. OSGi^{TM}- The Dynamic Module System for Java^{TM}. *http://www.osgi.org*.
20. Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massouli. Peer-to-Peer Membership Management for Gossip-based Protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.
21. Abhinadan Das, Indranil Gupta, and Ashish 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*, pages 303–312. IEEE Computer Society, June 2002.
22. Dawid Kurzyniec, Tomasz Wrzosek, Dominik Drzewiecki, and Vaidy Sunderam. Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters*, 13(2):273–290, 2003.

23. R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pages 256–267, February 2003.
24. Rich Wolski, Neil Spring, and Jim Hayes. Predicting the cpu availability of time-shared unix systems on the computational grid. In *Proc. of the the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 105–112, Washington, DC, USA, 1999.
25. Standard Performance Evaluation Corporation. Spec cpu2006 results, August 2008.
26. Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *SIGOPS Oper. Syst. Rev.*, 22(1):8–32, January 1988.
27. Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.
28. Hyperic Inc. SIGAR (System Information Gatherer and Reporter). *http://www.hyperic.com/products/sigar.html*.
29. Sun Microsystems Inc. Java Management Extensions (JMX) Remote API (JSR-160). *http://jcp.org/en/jsr/detail?id=160*.
30. Distributed Management Task Force Inc. Common Information Model (CIM) Standards. *http://www.dmtf.org/standards/cim/*.
31. Rich Wolski, Neil Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5-6):757–768, 1999.
32. Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.
33. J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.
34. Sasa Markovic. JRobin, a Java port of RRDTool. *https://rrd4j.dev.java.net*.
35. Cohesion Project Website. *http://www.cohesion.de*.
36. yWorks GmbH. yFiles. *http://www.yworks.com*.
37. David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 482–491, Washington, DC, USA, 2003. IEEE Computer Society.
38. Sun Microsystems Inc. Java Management Extensions (JMX) Specification (JSR-3). *http://jcp.org/en/jsr/detail?id=3*.