Vrije Universiteit Amsterdam
Faculty of Sciences

Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Michał Ejdys
Student number: 1544055 (VU), 201258 (WU)

A peer-to-peer Grid monitoring system

Master's Thesis
in COMPUTER SCIENCE
in the field of DISTRIBUTED SYSTEMS

Supervisors

Dr. Thilo Kielmann
Prof. Maarten van Steen
Vrije Universiteit Amsterdam

Dr. Janina Mincer-Daszkiewicz
Warsaw University

July 2006

Abstract

Grid monitoring systems aim at delivering accurate dynamic information about the Grid, which helps users with optimal utilization of the resources. Current systems have a static structure or are centralized services – thus are sensitive to nodes failures, not scalable, and difficult in administrating. In this work we propose an architecture of a peer-to-peer Grid monitoring system, addressing these issues. Furthermore, we propose three information disseminating strategies and evaluate their prototype implementations.

Contents

Chapter 1

Introduction

Grid monitoring systems provide meta-information about the observed environment. They monitor highly dynamic information, such as network bandwith, CPU load, job queue waiting time, etc. On the other hand, Grid information systems provide access to semi-static system characteristics. They are responsible for resource discovery, accounting and authorization.

Users need monitoring systems for optimal resource utilization. Information about load helps them with selecting the best resource available for the next task to be submitted on the Grid. Answering complex user queries may involve not only the monitoring system, but also resource discovery. In this work, we regard a *Grid monitoring system* as a system providing users with dynamic Grid information as well as with resource discovery services.

A main problem of existing monitoring systems is scalability. They are either centralized, thus introducing a single point of failure and a performance bottleneck, or have a static structure, making maintenance practically impossible in a large environment. Overcoming these weaknesses has motivated our work.

A natural answer to the scalability problem is a distributed system. Massie *et al.* in [9] identify *key design challenges* for distributed monitoring systems: scalability (up to thousands of nodes), robustness, manageability, portability, extensibility, and overhead.

To address scalability, robustness, and manageability, a structured peer-to-peer (P2P) network might be used. The decentralized nature of the P2P assures scalability. Additionally, the dynamic and self-organizing structure of P2P networks has minimal configuration requirements and is a solution to the remaining problems.

The cooperation between P2P networks and Grid computing, however, is still in its early stage. Although both technologies seem to have the same objective – coordinated use of large sets of distributed resources – they are being developed independently by different communities.

P2P and Grids have both the same general approach to fulfilling their main objective, i.e. utilizing the overlay structures. The overlays (virtual networks) coexist with underlying organizational structures (hardware and physical networks), but not necessarily correspond in structure to them. However, each has some crucial limitations – Grid computing focuses on infrastructure rather than failure, whereas P2P does quite the opposite: addresses failure but not yet infrastructure. Foster and Iamnitchi in [5] suggest that the complementary nature of the strengths and weaknesses of the two approaches will cause the interests of the two communities to grow closer over time.

The objective of our work is the design of a peer-to-peer Grid monitoring system. The proposed architecture uses an acyclic graph over P2P network, which allows for effective data aggregation and caching. Our system works in an event-driven fashion with measurements

and queries as events.

In our work, we discuss and evaluate three strategies for disseminating information: pull, push, and mixed. In the pull approach a user query is the event, causing the network to collect all the information needed for the answer. The push-based model reacts to measurements, trying to spread new values across the network and thus to be ready to immediately answer user queries. In the mixed approach we try to join aforementioned concepts to leverage their strengths and overcome weaknesses.

The rest of this thesis is structured as follows. Chapter 2 presents the study of related work, including existing Grid monitoring systems, as well as P2P networks. In Chapter 3 we describe in detail functional requirements for a Grid monitoring system. We present example user queries, their classification and required information to serve these queries. Chapter 4 describes the proposed architecture together with a detailed description of three approaches to information disseminating (pull, push, and mixed). Chapter 5 presents the prototype implementation and its evaluation. Finally, Chapter 6 concludes and suggests future work.

Chapter 2

Related work

In this chapter we present current projects and concepts in the field of Grid monitoring systems together with their strengths and weaknesses. We dedicate a separate section to SIENA – an interesting example of a large-scale event notification system. Finally, we present an overview of peer-to-peer networks and current achievements in the field of incorporating P2P concepts into Grids.

2.1. Grid monitoring systems

There exist quite a few monitoring systems for Grids. In this section we present the most important ones, not only describing their architecture, but also identifying their strengths and weaknesses. The study is the basis for determining functional requirements for Grid monitoring systems, described in detail in Chapter 3.

2.1.1. Ganglia

Ganglia [9, 14] is a distributed monitoring system with a hierarchical design. It works at two levels: clusters and federations of clusters (see Figure 2.1).

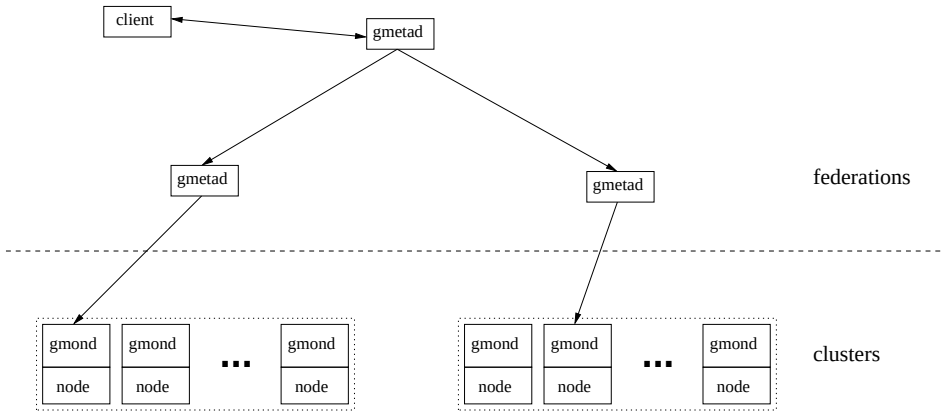


Figure 2.1: Ganglia architecture (adapted from [9])

There is a *Gmon* daemon present at each node within a cluster. *Gmon* monitors a node at which it is located, collecting 37 different measurements (e.g. number and speed of CPUs, their load, available memory, operating system). Moreover, daemons within a cluster communicate with each other using multicast over a local network to exchange collected information. As

a result, each node has complete information on the status of the entire cluster and can be regarded as its representative. This assures resilience to nodes failures. In case the current representative fails, any other node within the same cluster may be contacted by external processes.

Selected nodes run *Gmeta* daemons. They are organized in a tree of a static structure, defined by the system administrator. Static trees are used for simplicity, because the authors of Ganglia believe that Grids usually have only a small number of participating sites, even though the number of clusters might be large. Worse yet, Ganglia does not provide any automatic recovery from *Gmeta* daemons failures.

Each leaf node in the tree logically represents a single cluster. Each non-leaf node represents a federation of clusters. *Gmeta* periodically polls data from its children (*Gmons* and *Gmetas*). For children being clusters, it stores complete information about their measurements. For federations, it stores only summary information (sum and number of values for each measurement for all clusters in all federations in the subtree).

In Ganglia, only the root node answers queries. Consequently, the root is a performance bottleneck and a single point of failure. The query is processed by forwarding it to children. A node that knows the answer, does not pass the query to its children. Thus, queries are propagated potentially throughout the entire *Gmeta*-tree.

The system is designed to answer queries on the value of a measurement on a specific node, rather than ranking resources (e.g. finding the least loaded node).

2.1.2. Delphoi

Delphoi [8] is a centralized Grid monitoring system built from two types of components: a single query processing service (*Delphoi*) and its helpers (*Pythias*) – see Figure 2.2.

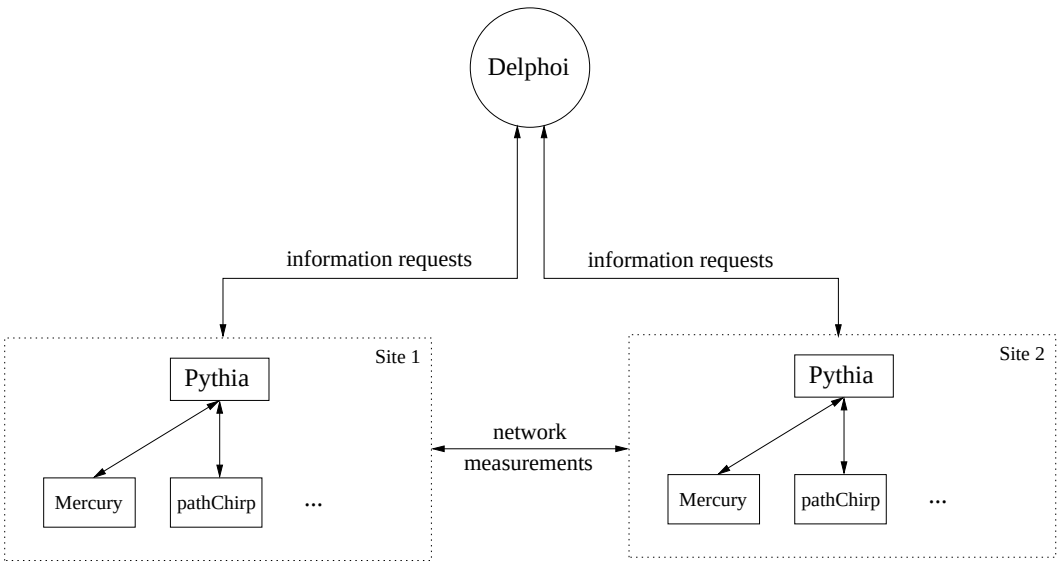


Figure 2.2: The Delphoi system architecture (adapted from [8])

The system provides the user with different kinds of information: meta data (active sites, known measurements), low-level resource and network information (delay, bandwith), high-level network information (optimal configuration of TCP connections, transfer time estimation), and queueing information (available queues, configuration, average waiting time and number of free nodes). Not only gives it current measurements, but it is also capable of

computing predictions. Its predictor is a version of the Network Weather Service (see Section 2.1.4) forecaster library.

Pythias are responsible for gathering information. They work by continuously collecting measurements from the computers they monitor. *Pythias* use specialized modules to communicate with external measurement providers (e.g. Mercury – a Grid monitor for resources and applications, or pathChirp – an estimator for available bandwith). Usually one *Pythia* controls one site.

The *Delphoi* service is responsible for answering all the queries. It handles low-level (e.g. CPU load, bandwith) and high-level queries (e.g. transfer time estimation). The answer to a low-level query is taken from an appropriate *Pythia* or retrieved from *Delphoi's* cache. For high-level queries, proper low-level queries are first answered. After performing some computations on received information, the final answer is returned.

Interesting use cases for the system are: selecting the best available replica, queue waiting time estimation, and transfer protocol optimization. However, a single *Delphoi* service is a performance bottleneck and a single point of failure.

2.1.3. CoMon

The main goal for CoMon [11] (*a mostly-scalable monitoring system for PlanetLab*) is to help administrators and users with tracking problematic machines on PlanetLab. Therefore, it contains special measurements for that specific environment (e.g. statistics of resource utilization per user account spread over a set of nodes). For details on PlanetLab, see Section 5.2.2.

The system has a centralized service that periodically (currently every 5 minutes) polls information from daemons running at nodes in PlanetLab. Additionaly, it answers user's queries. To be more precise: it presents the history of resource utilization, allowing for filtering the available data (e.g. showing only nodes with a load below given threshold).

CoMon is a centralized system. As a consequence, scalability becomes an issue. However, since it is mostly used for monitoring rather than efficient resource allocation, the information served to clients is allowed to be stale. Thus, whenever the number of monitored sites increases significantly, the frequency of gathering information at the centralized service is simply decreased.

2.1.4. Network Weather Service (NWS)

The aim of the Network Weather Service is to maximize four functional characteristics [16]:

- predictive accuracy (to provide accurate predictions in a timely manner),
- non-intrusiveness (not to introduce additional load on the existing system),
- execution longevity (to be able to operate logically indefinitely),
- ubiquity (to be accessible from all potential execution sites within a resource set).

The service is built out of four types of components, i.e. Sensor, Persistent State, Name Server and Forecaster (see Figure 2.3).

Sensors are located at monitored resources and gather measurements. Persistent State takes care of storing all the measurements. Many Sensors may use the same Persistent State. The Name Server is a centralized directory for binding names with TCP/IP addresses and discovering available sites. This is currently implemented with LDAP. Finally, Forecaster

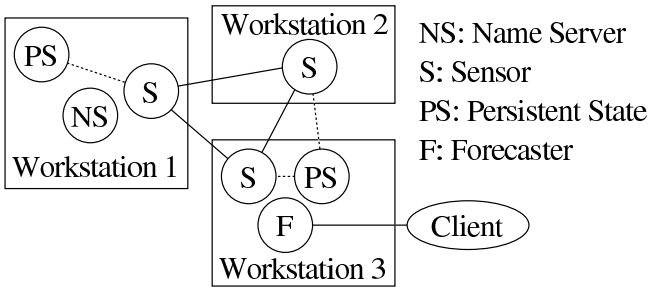


Figure 2.3: NWS processes distributed across three workstations (source: [16])

makes predictions for a given resource and a given time frame. This is the only process clients contact.

The emphasis of the project is on collecting measurements and making predictions. For example, the system reduces the number of point-to-point network measurements (e.g. latency) by organizing sensors in a hierarchy of cliques. A clique is simply a group of nodes physically close to each other (e.g. within the same campus). Thus, network measurements with distant sites can be computed once for the entire clique and then approximated for its members.

Two main weaknesses of the system are: the manually managed structure of connections between sensors and a centralized Name Server.

2.2. Scalable Event Notification Service (SIENA)

The Scalable Event Notification Service [2] is an interesting example of content-based networking. It is an event-driven model of event dispatching.

The dispatching is regulated by *advertisements*, *subscriptions* and *notifications*. Objects of interest specify the events they intend to publish by means of *advertisements*. Interested parties specify the events they are interested in by means of *subscriptions*. Objects of interest can then publish *notifications* and the event service takes care of delivering them to the interested parties.

An *event* consists of its identifier, timestamp and a set of attributes. An *event filter* defines a class of event notifications by specifying a set of attribute names and types and some constraints on their values. A *pattern* is defined by combining a set of event filters using filter combinators (or, and). An event notification is delivered to a subscriber only if its subscription pattern matches the given event.

In SIENA, nodes are organized either in a hierarchical or peer-to-peer topology. SIENA introduces two algorithms for constructing notification paths for events: subscription and advertisement forwarding.

The first approach uses subscriptions to set paths for notifications. Every subscription is stored and forwarded from the originating server to all the servers in the network. As a result, a tree that connects the subscriber with all the servers is formed. Whenever an object publishes a notification that matches that subscription, it is routed towards the subscriber following the reversed path established by the subscription.

Advertisement forwarding uses advertisements to set the paths for subscriptions, which in turn sets the paths for notifications. Every advertisement is forwarded throughout the network. As a result, a tree that reaches every server in the network is formed. Whenever a server receives a subscription, it propagates the subscription in reverse along the path to the

advertiser, thereby activating the path. Notifications are then forwarded only through the activated paths.

Both algorithms result in a minimal spanning tree for each source.

2.3. P2P networks

Peer-to-peer networks are self-organizing distributed systems without any hierarchical structure or centralized control, overlayed on IP networks. Among their most important features are robust routing, scalability and fault tolerance. They fall into two categories [7]: unstructured and structured networks.

2.3.1. Unstructured P2P

Unstructured peer-to-peer networks organize peers in a random graph and use flooding, random walks or expanding-ring Time-To-Live (TTL) search on the graph to query content stored by peers. Each peer evaluates the query locally on its own content. This is inefficient because queries for content that is not widely replicated must be sent to a large fraction of peers. Additionally, there is no coupling between topology and a data item's location.

Most interesting examples of unstructured P2P networks are: FreeNet, Napster, BitTorent, and Gnutella. The latter will be described in detail.

Peers in Gnutella perform actions of both servers and clients (and they are named accordingly: *serv*ers+cli*ents* = *servents*). Each peer participates in maintaining the network, forwarding, and answering users queries. Servents exchange the following messages: *ping* (for manifesting their presence), *pong* (a response to ping, containing information about existing peers), *query* (a user specified search string), and *query response* (containing information necessary to download a file).

Gnutella uses broadcasting within a limited scope for ping and query messages. They are identified by a randomly generated id, which prevents rebroadcasting messages. Limited scope of broadcasting is done by utilizing TTL counters, decreased at each hop. Answers to broadcast messages (pong to ping and query response to query) are back propagated via paths established by the original messages.

In order to join the network, each new peer has to contact one of the hosts known to be on-line most of the time. Their list is usually published on a web-site. The new peer starts operating by broadcasting a ping message via hosts from the acquired list. As a response, it receives the list of some other peers in the network. After joining the network, nodes periodically exchange ping and pong messages to maintain network connections. This mechanism assures network integrity and makes Gnutella failure robust.

How the user queries are broadcast can be seen in Figure 2.4. Each peer receiveing a query message tries to match it to the resources it stores. Furthermore, it forwards the message to its peers. Answers to queries contain contact information to hosts storing data items the user is searching for. After receiving the answer, a user directly contacts appropriate hosts by exchanging *get* and *push* messages.

Because of locality of query processing (flooding within limited scope), searching is effective only for widely spread content.

2.3.2. Structured P2P

Structured peer-to-peer network assigns keys to data items and organizes its peers into a graph that maps each data key to a peer. This structured graph enables efficient discovery of

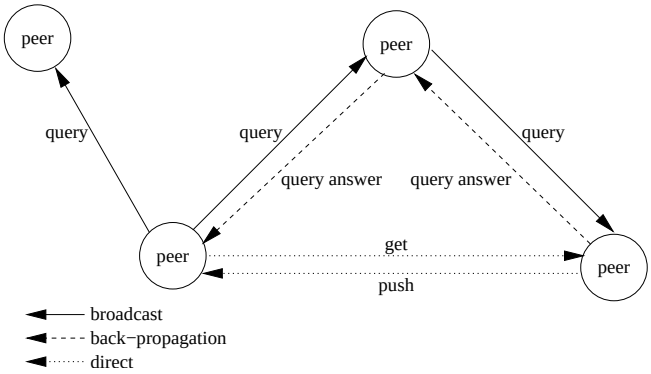


Figure 2.4: Queries in Gnutella

data items using the given keys. Examples include Content Addressable Network (CAN) [12], Chord [15] and Pastry [13]. The latter will be described in detail in the next section together with an application-level multiast overlay, Scribe.

2.3.3. Pastry and Scribe

Pastry

Pastry [13] is a scalable, self-organizing peer-to-peer location and routing substrate. Each node in the Pastry network has a unique numeric identifier (nodeId). The nodeId is assigned randomly when a node joins the system. It is assumed that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space.

For the purpose of routing, nodeIds and keys are represented as a sequence of digits with base 2^b (b is a configuration parameter with typical value 4). Pastry routes messages to the peer whose nodeId is numerically closest to the given key. In each routing step, the message is forwarded to a node whose nodeId shares with the key a prefix that is at least one 'digit' (b bits) longer than with the present nodeId. If no such node is known, the message is forwarded to a peer with common prefix of the same length, but numerically closer to the message key. For an example, see Figure 2.5(a).

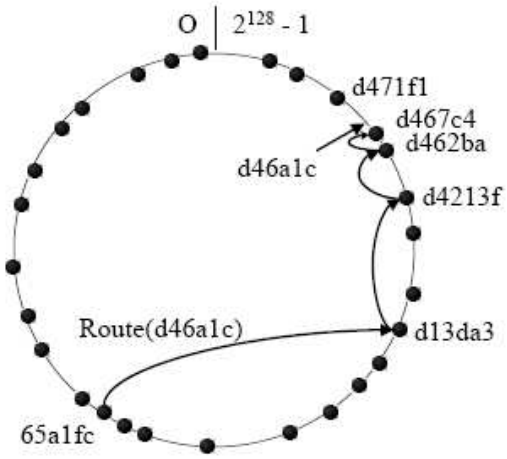
Assuming a network consists of N nodes, Pastry can route messages in less than $\lceil log_{2^b} N \rceil$ steps under normal operation.

In order to route messages, a *routing table* is maintained at each node – see Figure 2.5(b). It contains nodes for prefixes of different length. Additionally, a *neighborhood set* with M (being a parameter) closest nodes is being kept. The set is used only for optimizing the routing table, not for routing itself.

The neighborhood set is also used to assure network locality. It aims at minimizing the distance messages travel, according to a scalar proximity metric provided by the application (e.g. the number of IP routing hops). This is achieved by always selecting the closest known node as representative for the each prefix. More precisely, a message is forwarded to a relatively close node with a nodeId that shares a longer common prefix or is numerically closer to the key than the local node.

Scribe

Scribe [3] is a large-scale, decentralized application-level multicast infrastructure built on Pastry. It allows for creating topic groups and then publishing messages to subscribers.



(a) Routing a message from node $65a1fc$ with key $d46a1c$. The dots depict live nodes in Pastry's circular namespace

(b) Routing table of a Pastry node with nodeId $65a1x$, $b = 4$. Digits are in base 16, x represents an arbitrary suffix

Figure 2.5: Routing messages in Pastry (source: [13])

Any Scribe node may create a group. Other nodes can then join the group and multicast messages to all members of the group. Each group has a unique groupId. The Scribe node with a nodeId numerically closest to the groupId acts as the rendez-vous point for the associated group. The rendez-vous point is the root of the multicast tree created for the group.

To create a group, a Scribe node asks Pastry to route a *create* message using the groupId as the key. Pastry delivers this message to the node with the nodeId numerically closest to groupId, similarly for a *join* request message. Each node through which a *create* or *join* message is routed, becomes a forwarder for the given group, maintaining children table with all nodes that passed the message to it. As a result, a multicast tree is formed, spanning all members of the given group.

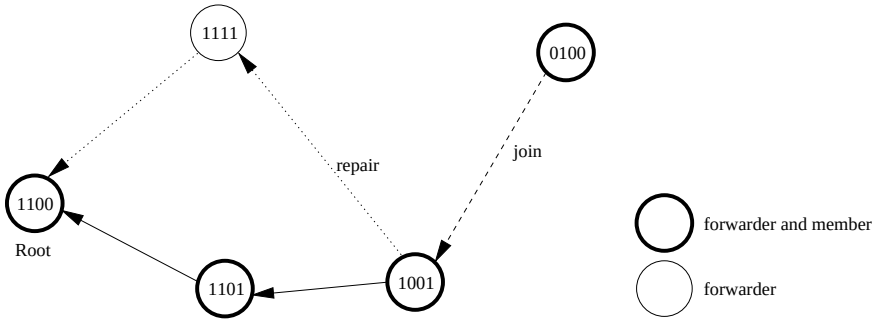


Figure 2.6: Tree maintenance in Scribe (adapted from: [3])

An example of tree maintenance procedures is shown in Figure 2.6. Node 1100 is the rendez-vous point for the example group. Nodes 1101 and 1001 are group members and – at the same time – forwarders. The joining node, 0100, sends the join message to the group id. Pastry routes this message to node 1001 in the first step. Upon receiving the join message, 1001 registers 0100 as its child. This is needed for proper dissemination of group messages. Since 1001 is already a forwarder for the group, it is part of the multicast tree. Consequently, this join message is not forwarded and the join procedure for 0100 ends.

Scribe periodically monitors its connections and whenever a node failure is discovered, the system reacts by reconstructing the multicast tree. For example, if node 1001 discovers that 1101 is down, it asks Pastry to route a join message so that a new connection in the multicast tree can be established. Pastry routes the join message to node 1111. Since that node was not part of the tree, not only it registers the fact that node 1001 is its child, but also forwards the join message further towards group id. Next node on the way is the root node. It registers node 1111 as its child.

Failures of the rendez-vous points are handled similarly. As soon as any of the root's children discovers that rendez-vous point is no longer available, it routes the join message. Since Pastry delivers messages to the nearest (with respect to ids) available node, new root node is selected and notified automagically.

The described method of building and maintaining the multicast tree limits the number of nodes involved in each action. All the tree repairs are performed locally (e.g. node 0100 does not have to be notified about changes in the tree above its parent).

It is worth mentioning that periodically sent messages not only help with repairing the tree, but also with optimizing the tree whenever a new node appears in the network. If a new node happens to be a better peer (in terms of locality and number of routing steps), the tree is reconstructed accordingly. Thus, Scribe benefits from Pastry's locality properties.

Messages publishing in Scribe is done by sending the message to the root node of the multicast tree. As soon as Pastry delivers the message to the rendez-vous point, it is resent to its children. Similarly, every node in the tree handles the received message.

2.4. P2P and Grids

Current Grid systems have two serious problems: the scalability and the disability to handle node failures. Both problems are caused by centralized services, which are mostly needed for resource discovery. As Foster and Iamnitchi suggested in [5], convergence of Grid computing and peer-to-peer networks seems to be a natural step. Enriching Grids with abilities of P2P networks would solve scalability issues and help with handling nodes failures. Additionally, P2P could also provide Grids with a decentralized way of discovering available resources.

Surprisingly, there exist only a few projects that join experiences from both fields. Examples are Zorilla [4], the Web-Services Discovery Architecture (WSDA, [6]) and NaradaBrokering [10].

Among them, the most interesting project is Zorilla – a large-scale job scheduling system. In order to benefit from P2P networks, its authors completely redesigned the concept of Grid computing, instead of extending it. Zorilla does not need a centralized service to submit a job, nor requires synchronization of many independent job schedulers. It simply uses flooding to publish the job request locally.

Instead of making a globally optimal decision about job scheduling, Zorilla exploits locality and benefits from the scalability of the solution. The job is simply published within a certain scope from the user node.

Chapter 3

Queries and information in Grid monitoring systems

The starting point for designing our system is collecting requirements. After studying the systems presented in Section 2.1 and their example applications, we identified the most common types of queries that Grid users actually need and use. In this chapter we also present information needed to handle these queries and possible strategies of aggregating them.

3.1. Queries

Possible queries that could be of Grid users' interest, fall into two categories: *direct measurements* and *resource discovery*.

Direct measurements provide users with detailed information about specific resources – their utilization, hardware and software specification, etc. This helps with verifying that a given resource is suitable for an application (e.g. it has sufficient memory available or the CPU's frequency is high enough). Additionally, this category includes host-to-host network measurements, which are useful for efficient usage of available resources – e.g. by allowing selection of the best replica to use.

Resource discovery queries in their simplest form return a list of all available resources. More advanced queries allow users to express some requirements for the resources (e.g. minimum available memory, type and version of the operating system) or some preferences (e.g. job queue average waiting time or CPU load below some value). Additionally, a user may request the system to rank the resources accordingly to their accessibility, e.g. by CPU load or job queue waiting time.

Resource discovery queries could be further classified based on the presence of constraints. *Queries with constraints* allow for retrieving the list of resources meeting the specified requirements (e.g. minimum memory available). *Queries without criteria*, in turn, allow only for retrieving the list of *all* resources.

Unfortunately, current systems do not offer advanced resource discovery. For example in Ganglia, Delphoi or NWS (see Section 2.1) the user can only acquire the list of *all* available resources. In order to rank the resources or filter out unsuitable ones, the user is forced to submit a series of additional direct measurements queries.

Since we believe that resource discovery is of highest importance to the user and direct measurements can be easily implemented in a decentralized manner, we concentrate our efforts on advanced resource discovery services. Surely, the system should also offer direct measurements. They, however, can be regarded as complementary information.

3.2. Information

For the sake of answering queries described in the previous section (*direct measurements* and *resource discovery*), a Grid monitoring system has to collect the following information:

- node measurements – total and available disk space, swap space and memory; load and frequency of the CPU,
- job queues – their configuration (number of CPUs, job manager used, maximum allowed jobs, maximum memory available) and utilization (average waiting time, average waiting jobs),
- network measurements – low-level metrics like latency, bandwith (available and utilized), path (in terms of IP hops) and high-level estimates deducted from them, e.g. expected transfer time between hosts or optimal TCP options.

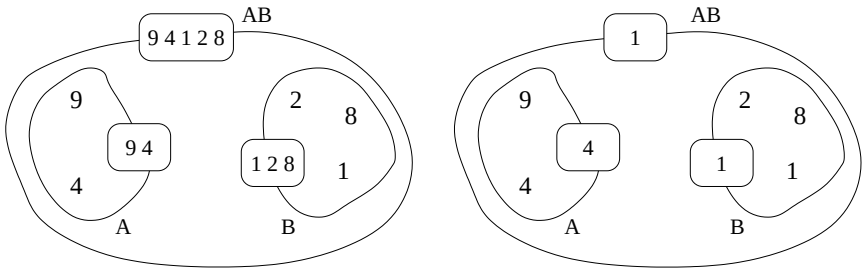
In the system we are proposing, the emphasis is put on node measurements. Information about job queues is very similar to node measurements and thus can be designed and implemented similarly. Network measurements, in turn, are omitted, since they are only used in the already excluded *direct measurements* category.

Information coming from nodes becomes a basis for both ranking resources (e.g. by CPU load) and filtering using criteria (e.g. by available memory).

3.3. Aggregating information

Besides collecting and serving the information itself, a Grid monitoring system can operate on its aggregated form. *Aggregated information* presents an extract of information. It allows for answering some queries without investigating all information in detail. Possible types of aggregation in our system are: *union*, *minimum*, and *maximum*.

Union presents joint information about all elements of a set. Keeping that kind of aggregated information might enable a quick access to the complete list of all resources. Figure 3.1(a) presents the example.



(a) union (b) minimum

Figure 3.1: Examples of aggregated information

Minimum and maximum can be useful to quickly determine the 'best' element (i.e. with highest or lowest value of some measurement) and to optimize finding answers to queries with constraints.

Whenever a query with constraints is analyzed by the system, aggregated information about some set of resources can help with deciding whether the particular set should be

analyzed in detail or not. This technique can significantly limit the number of resources being involved in answering the query.

Figure 3.1(b) presents an example together with aggregated information (minimum). In this case, the answer to the query 'return all values below 3' could be computed as follows. At first, we decide to look into the set AB – its minimum value (*1*) assures that there are suitable elements inside. Further, since A has minimum value of 4, it should be skipped – all its elements would not meet the constraint 'below 3'. B, however, has to be investigated in detail, contributing values *1* and *2* to the answer.

3.4. Predictions

Some Grid monitoring systems (like NWS or Delphoi) offer predictions in addition to current measurements. The motivation is that for users it might not be important what the load of the machine *is* or *was*, but – what the load *will be* when they decide to use it. Consequently, all identified queries may be related either to the presence or to the future.

However, enriching existing system with a predictions module is not complicated. For example, the Delphoi system adapted the NWS forecaster library in order to compute predictions about job queues utilization.

In order to serve current information as well as calculated predictions for the limited time slots, our system will be able to operate on series of values. Forecasting will be performed by each node individually and only the results will be presented to other nodes.

Chapter 4

System architecture

After describing all the requirements, we are ready to propose the architecture of a peer-to-peer Grid monitoring system. In this chapter we present the architecture overview together with three strategies of disseminating information.

4.1. Architecture overview

We propose to build the Grid monitoring system on top of a peer-to-peer network in order to obtain a decentralized self-organizing structure. With minimal configuration and administration effort, a scalable system can be maintained.

We assume that an acyclic graph of peers can be created on top of a P2P overlay (as our prototype implementation proves, this can be easily achieved) – see Figure 4.1.

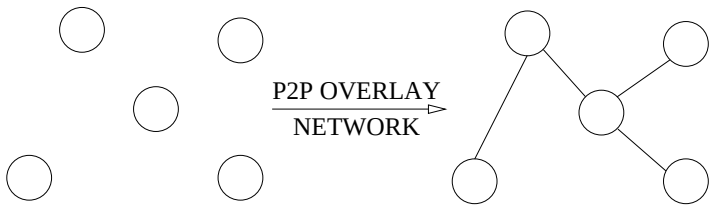


Figure 4.1: P2P overlay network

The Grid monitoring system resides on top of a P2P overlay network – see Figure 4.2. The application processes client queries and manages the infrastructure for optimizing the processing. It is a mediator between the user and the measurements. The application uses the P2P overlay network to communicate with all the nodes.

The user may contact any of the nodes in the network to submit a query, since the application treats all the nodes equally and exports the same interface to each of them.

For query processing and information disseminating, we propose the following three strategies:

1. pull approach,

   All user queries are answered by propagating them throughout the entire network and gathering information. However, no additional communication is required.

2. push approach,

   Each node maintains a cache for measurement, keeping aggregated information about the rest of the network. Thus, user queries (without constraints) are answered locally.

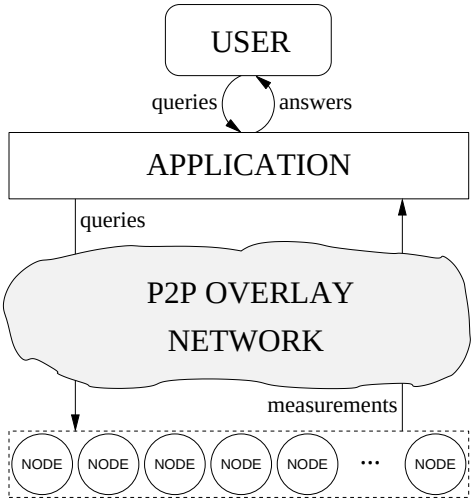


Figure 4.2: System architecture

However, in order to maintain the cache, peers need to constantly exchange information about new measurements.

3. mixed approach.

   This approach is a combination of the aforementioned two solutions. Some nodes of the network work in push mode, others in pull mode.

These three strategies will now be discussed in detail.

4.2. Pull approach

In the pull approach the query is the event, which causes messages to be propagated. The only communication is directly caused by the submitted query. There is no information cached in this approach. Figure 4.3 presents an example graph of 5 nodes (A, B, C, D, and E) with their current measurement value (7, 5, 1, 2, and 3 respectively).

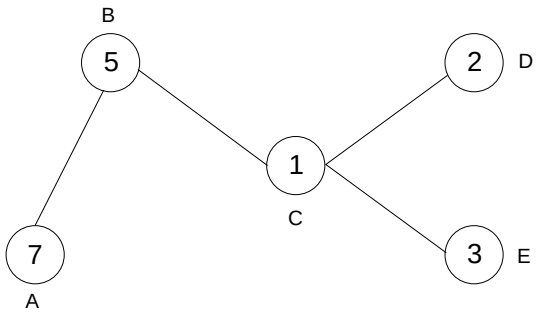
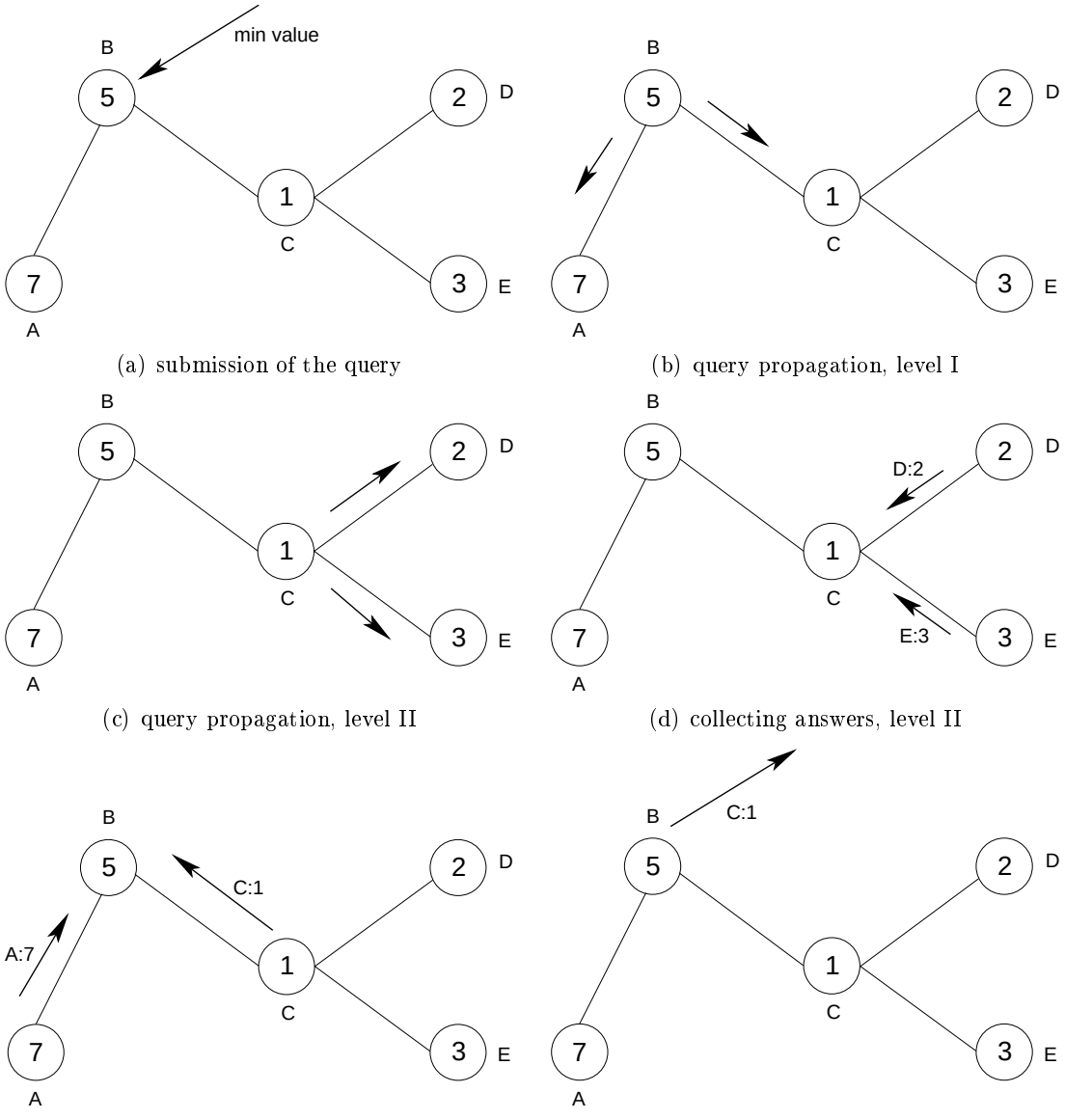


Figure 4.3: Example graph in pull approach

Regardless of its type (with or without constraints), queries are processed in the same way. Whenever a user submits one, information is gathered from the entire network.

Figure 4.4 presents the process of answering an example query without constraints (obtaining the minimum value in the network). At first the query is submitted to node B – see



(a) submission of the query

(b) query propagation, level I

(c) query propagation, level II

(d) collecting answers, level II

(e) collecting answers, level I

(f) returning the answer

Figure 4.4: Query without constraints in pull approach

Figure 4.4(a). In Figures 4.4(b) and 4.4(c) it is propagated throughout the network. In Figures 4.4(d) and 4.4(e) the answer is returned. Node C calculates the minimum value based on the answers it received from its peers (*2 at D* and *3 at E*) and its own value (*1*). Consequently, the answer it passes to B is *1 at C*. Likewise, node B computes the minimum of its own and received values, returning *1 at C* to the user – see Figure 4.4(f).

In the pull approach network traffic is generated only when a query is submitted. Intuitively, the trade-off will be the quality of information. In our work, the quality of information is measured in terms of *staleness*. Its definition will be formulated in Section 5.2.4. Roughly, staleness is the time passed since the acquired information has stopped being correct.

In the case of the pull approach, staleness is expected to be proportional to the *maximum* network latency between a node receiving a user's query and all other nodes. The reason is that the node being queried initiates pulling and has to wait for all the answers to come.

This approach is expected to be suitable for systems with a low query load, i.e. when maintaining a cache for measurements (as in push and mixed models) would not pay off. We will return to this issue at the end of the next section.

4.3. Push approach

In the push approach a measurement is the event causing the exchange of information beetween nodes. They maintain a cache for values, which contains the aggregated information about each subnetwork represented by its peer nodes.

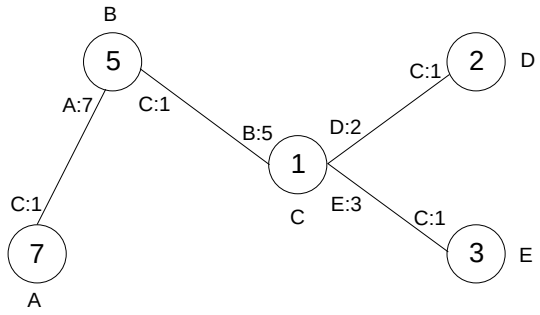


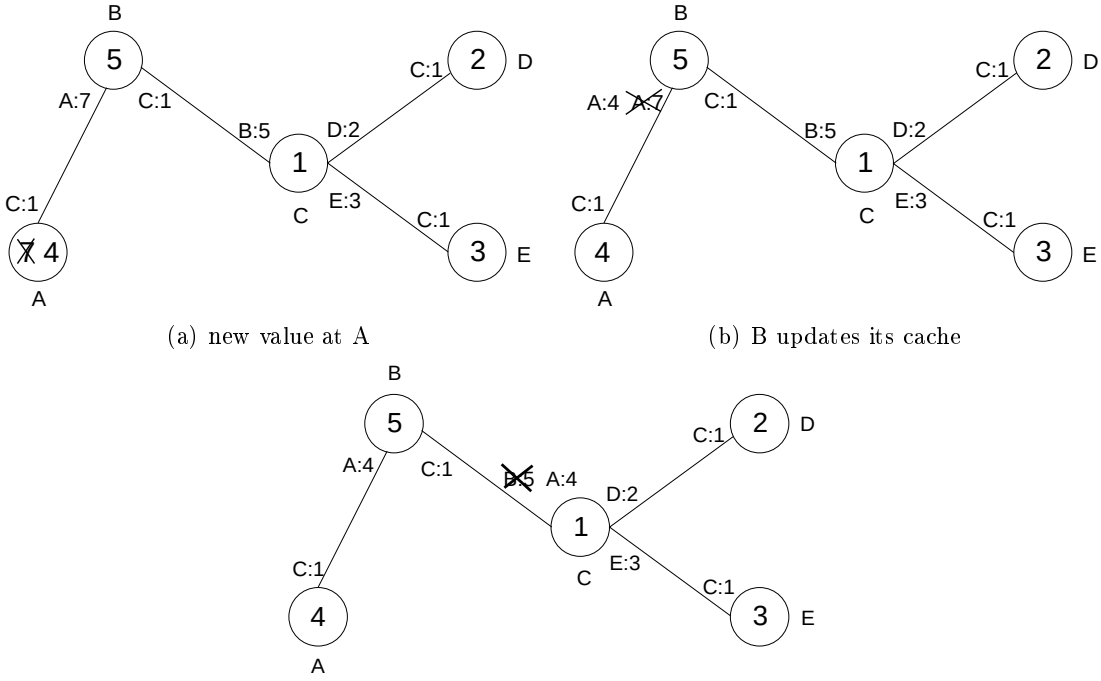
Figure 4.5: Minimum-cache in push approach

Figure 4.5 presents an example graph of 5 nodes with a cache that aggregates information by calculating a minimum. Each node keeps aggregated information for every connection it has. For example node A has only one peer-node, i.e. node B. It represents the subnetwork consisting of nodes B, C, D and E. Among them, node C has a minimum value of 1. Consequently, node A stores information *min of 1 at C* (*C:1*) for its connection to node B.

Similarly, node B stores cache information for its two peers: A and C. The connection with A represents a one node subnetwork. Thus, aggregated information for it is *A:7*. Node C represents the subnetwork consisting of nodes C, D and E with a minimum value of 1 at node C. Thus, a cache value for B's connection with C is *C:1*.

Whenever a new value arrives at a node, it recalculates the aggregated information it should present to all its peers and – for which an update is needed – sends a notification.

Figure 4.6 shows changes after a new value of 4 appears at node A. At first, node A updates its local information – see Figure 4.6(a). Then, A updates information it has presented to node B since it became outdated. New information (*min of 4 at A*) is sent to B. Node B updates its cache – see Figure 4.6(b). Since the new value is below the minimum value presented by



(a) new value at A

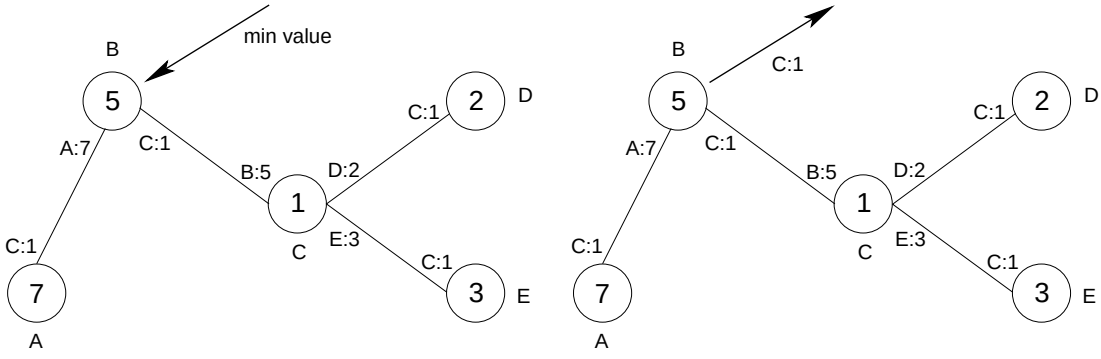
(b) B updates its cache

(c) C updates its cache

Figure 4.6: Updating cache in push approach

B to C, the latter has to be notified. B informs C about the new value (*min of 4 at A*) – see Figure 4.6(c). However, information C presented earlier to D and E is still valid (*C:1*). Thus, the update process does not go beyond node C.

In the push approach, queries without constraints can be quickly answered – directly by the queried node. Figure 4.7 presents an example query submitted to node B about the current minimum value. Node B quickly calculates the minimum of aggregated information from its peers (*1 at C*, *7 at A*) and its own value (*5*) and returns the answer (*min of 1 at C*).



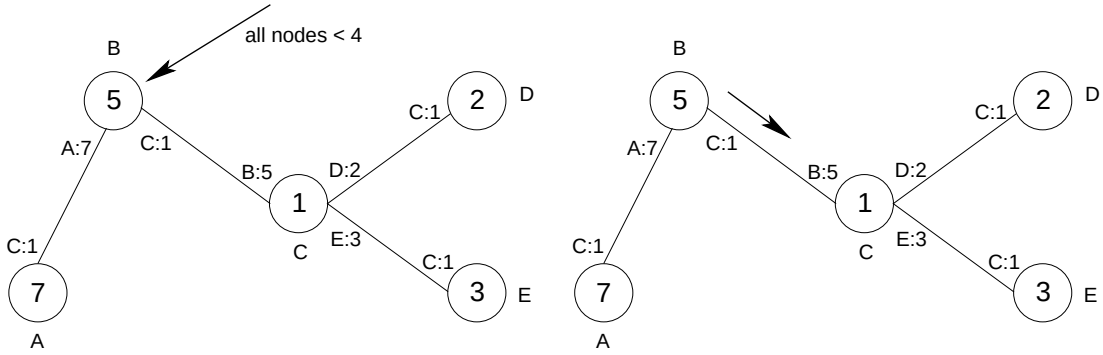
(a) submission of the query

(b) returning the answer

Figure 4.7: Query without constraints in push approach

Queries with constraints have to be propagated among peers. However, the query is forwarded only to these subnetworks, which aggregated information meets the requirements specified in the constraints.

Figure 4.8 shows an example query about all nodes with a measured value below 4. The



(a) submission of the query

(b) query propagation, level I

(c) query propagation, level II

(d) collecting answers, level II

(e) collecting answers, level I

(f) returning the answer

Figure 4.8: Query with constraints in push approach

example usage is a request to retrieve a list of all the nodes with average job queue waiting time below 4 minutes. The user submits a query to node B – 4.8(a). In the first step, node B decides it can skip the subnetwork represented by node A, since its minimum value is 7, meaning none of the nodes in this part of the network can satisfy the user's requirements. The query, however, is propagated to the other branch, where the minimum value is 1 – see Figure 4.8(b).

In Figure 4.8(c) node C forwards the query to both its subnetworks, since both can possibly satisfy the requirements. In Figure 4.8(d) nodes D and E return their answers with singletons containing only themselves.

Figure 4.8(e) presents the answer that node C sends back to B. It contains nodes that C received from its peers (D and E) and C itself, since it also satisfies the user's requirement. In Figure 4.8(f) B returns the answer to the user. B itself does not contribute to the answer, since its value (5) is greater then the user's threshold.

In this example, the list of resources was ranked accordingly to the measurement involved in the constraints, but a different measurement for ranking can be used. For example, retrieving a list of resources with average job waiting time below 4 minutes ranked by the amount of available memory is also possible,

Also limiting the list to a specified number of resources can easily be implemented by truncating the list at each step. For example, if the user requests only the top two nodes, node C in the stage presented by Figure 4.8(e) would return the list {*C:1*, *D:2*}.

Normally, the cache is constantly updated and queries are submitted randomly. Therefore, the quality of information at a specific node is expected to be proportional to the *average* (as opposed to *maximum* in the pull approach) network latency between the given node and all other nodes in the network.

The push approach, at the price of frequent updates needed to maintain the cache, gives a quick response to queries without constraints and limits the number of the nodes involved in processing queries with constraints.

The approach presented in this section is expected to be efficient in environments where the average number of queries per second significantly exceeds the average update rate of measurements at nodes. In that case the overhead of maintaining the cache pays off.

4.4. Mixed approach

The mixed approach merges the concepts of the push and pull approaches. Both queries and measurements initiate communication.

Leaf nodes (i.e. nodes with only one peer) work in push mode. Whenever a new measurement is done, information is propagated to a node's peers. Other, non-leaf nodes (i.e. having more than one peer) work in pull mode. They do not propagate information about subnetworks to other peers, but cache information about their leaf-peers.

Figure 4.9 presents an example network configuration and cache content in the mixed approach. A, D and E are leaf nodes working in push mode. They present information about their one-node subnetworks to their peers (A to B, D and E to C). Nodes B and C work in pull mode and thus do not exchange information with each other.

It is worth mentioning that nodes are not statically assigned to push or pull areas. In a real environment a graph of connections between nodes changes as nodes join and leave the network. Additionally, the next time the same peer joins the network, it may receive a different id, which influences its placements in the overlay network. Nevertheless, some nodes may always be leaf nodes, resulting in an increased (relatively to pull nodes) network traffic.

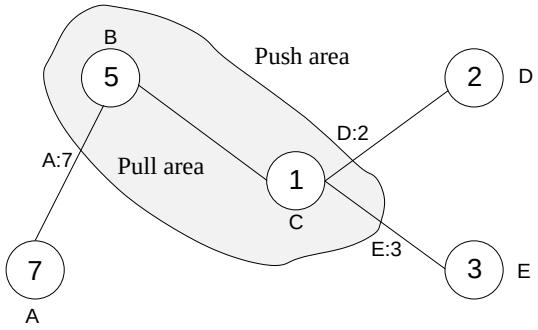
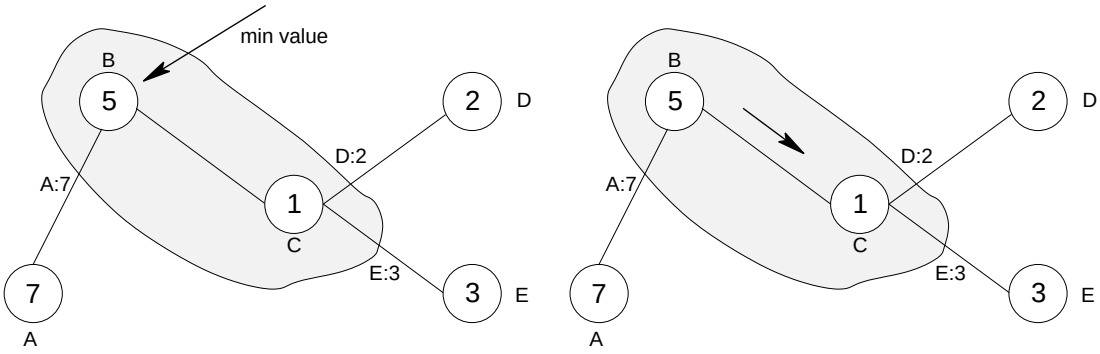
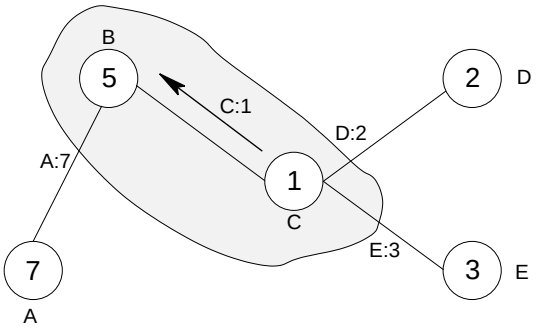


Figure 4.9: Minimum-cache in mixed approach

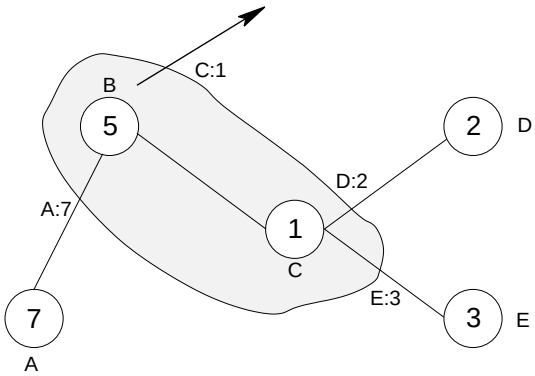


(a) submission of the query

(b) query propagation



(c) collecting answers



(d) returning the answer

Figure 4.10: Query without constraints in mixed approach

Queries with and without constraints are all processed in a way similar to the pull approach. A node propagates the query to all its peers within the pull area. For the remaining peers (i.e. peers from the push area) cached information is used.

Figure 4.10 presents an example query realization. Node B, after accepting the query (see Figure 4.10(a)), forwards it to node C, its only peer within the pull area (Figure 4.10(b)). Node C returns the answer immediately, as it has information about all its peers cached. The answer (*min of 1 at C*) is sent back to B (Figure 4.10(c)). Next, node B computes the minimum of: received value from C, cached value about A and its own value. Finally, it returns the answer (*min of 1 at C*) to the user (Figure 4.10(d)).

The mixed model reduces communication overhead of the push model, increasing performance (both query time and quality of answers) of the pull model at the same time. The quality of information in this approach is expected to be better than in the pull approach, but worse than in the push model.

Chapter 5

Evaluation

In addition to proposing a P2P Grid monitoring system architecture, we evaluated its prototype implementation. In this chapter we describe this implementation together with the testbed we used. We present and discuss results of the conducted tests.

5.1. Prototype implementation

For the prototype implementation of the proposed architecture, Pastry and Scribe (see Section 2.3.3) have been used. For an overview of the system design – see Figure 5.1.

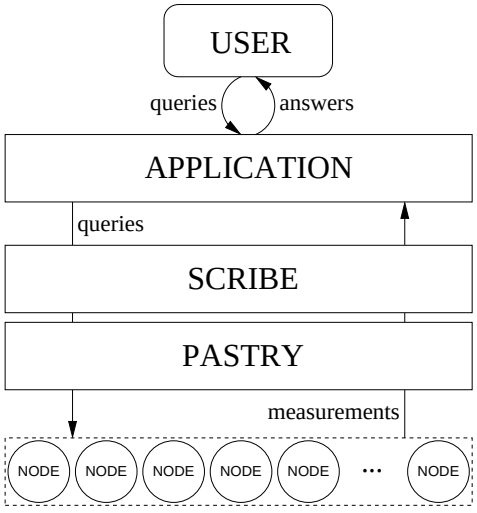


Figure 5.1: Prototype implementation

All three layers of the system (Pastry, Scribe, and application) will now be described in detail.

Pastry

Pastry provides an infrastructure for organizing all the nodes into a network. It handles nodes joining and leaving the network, and thus keeps instability of the environment under control. In addition, Pastry routes messages, supporting any kind of communication between nodes.

When using Pastry (or any other P2P without any centralized service), the bootstrapping issue arises. A new node starting its operation needs to know whom should it contact in order to become part of the overlay network. To solve this problem, we create a *bootstrap group*. It consists of nodes expected to be on-line most of the time. Their network addresses are registered under a single well-known name, using DNS.

While initiating, nodes retrieve the bootstrap group from DNS and try to join the overlay network, using one of the obtained addresses. In case none of the peers can be contacted, the node (given that it is a member of the bootstrap group), initiates a new Pastry network. Other nodes have to wait for one of the bootstrap peers to become reachable.

We assume that it is relatively easy to identify at least a few stable nodes within a group of interested hosts. However, it is not required for *all the hosts* from the list to be on-line *all the time*. Failure of all but one will still provide joining nodes with needed service. Failure of all will only disable new nodes from joining, but will not destroy the existing network.

Additionally, maintaining the bootstrap group requires minimal administrative work – only the DNS entry has to be modified, in case some nodes need to be replaced, removed or added.

Another advantage of using round-robin DNS scheme is load balancing [1]. Each time a DNS-server is queried, it will return a reordered list, causing successive nodes to be directed to different bootstrap peers.

However, DNS servers are not obliged by the standard to implement round-robin in the described manner. Servers may return the same list each time they are queried or reduce the list to one of its elements.

Scribe

The middle layer of our system is Scribe. It uses Pastry's message passing API to establish and maintain a multicast tree. Each of the nodes subscribes to the same topic and thus the Scribe tree spans over the entire overlay network.

Scribe introduces a hierarchy into the network. In our architecture, however, the tree is used to structure the network rather than to benefit from the parent-child relations. All the nodes are treated equally, as peers. Scribe offers simply one of the easiest ways to overlay the network with an acyclic graph – see Figure 5.2.

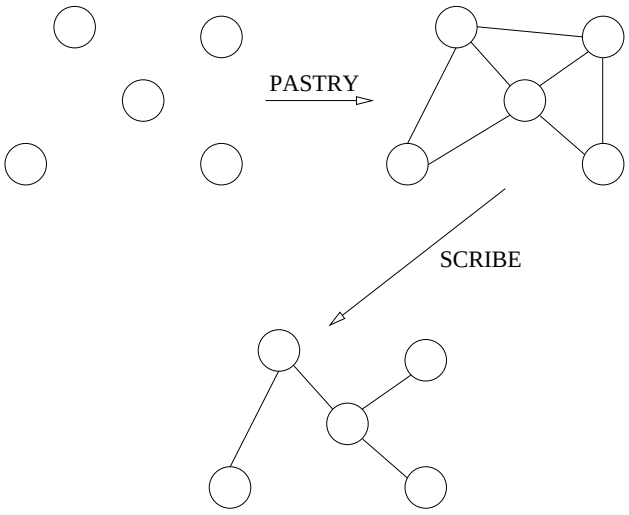


Figure 5.2: Role of Pastry and Scribe in the prototype implementation

Application

The application is laid on top of Scribe and is responsible for processing client queries. It also maintains the infrastructure for optimizing the processing and is responsible for implementing the three processing strategies.

Whenever the user submits a query, the application collects appropriate measurements from the nodes. It uses the intermediate layers (Scribe and Pastry) to communicate with the nodes in the network. As soon as all the information is collected, the application returns the answer to the user.

5.2. Tests setup

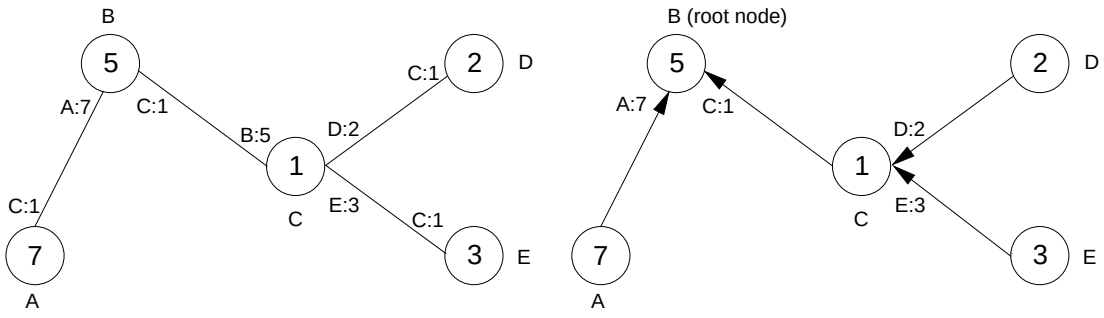
In this section we describe the testbed used in the evaluation. We start with explaining what simplifications to the architecture described in the previous section were made. Then we discuss PlanetLab – the environment we used. The section continues with the description of our efforts to stabilize this extremely unstable environment, followed by the definition of the quality of information – the basis for comparing different models. Finally, we present the methodology of our tests.

5.2.1. Simplifications

In order to establish a framework for comparing different approaches to query processing, several assumptions and simplifications have been made. We decreased complexity of the network structure and concentrated on queries of type minimum only.

The network structure

The most important simplification of the system architecture is reducing the graph to the tree and choosing only the root node to answer the queries. It significantly simplifies the implementation of approaches that use a cache (push and mixed). The node, instead of calculating aggregated information for each of its peers, has to present the information about its subtree to the parent node only – see Figure 5.3.



(a) network of peers (b) tree

Figure 5.3: Simplification of the network structure

Arrows connect children to their parents. In this example, B has been selected as a root node. Children present aggregated information only to their parents. For example, node C sends information about minimum in its subtree (consisting of nodes C, D, and E) to its parent – node B. Consequently, each node caches information not about every subnetwork,

it is connected to, but only about subtrees represented by its children. For example, node C caches information from D and E, but does not have any information from its parent, node B.

The parent-children relationship is retrieved directly from Scribe's multicast tree.

Minimum only

It is worth mentioning that minimum and maximum aggregation are very similar and thus one of them can be omitted. Additionally, we prefer minimum over union, since we expect it to reveal most significant differences between the presented approaches.

Therefore, from the information aggregation methods described in Section 3.3, we chose only minimum to be implemented and omit maximum and union.

5.2.2. Environment

We have chosen PlanetLab as a testbed for evaluating our system. PlanetLab is a planetary-scale network, currently consisting of 685 machines in 332 sites spread over 25 countries. See Figure 5.4 for the visualization of nodes distribution.

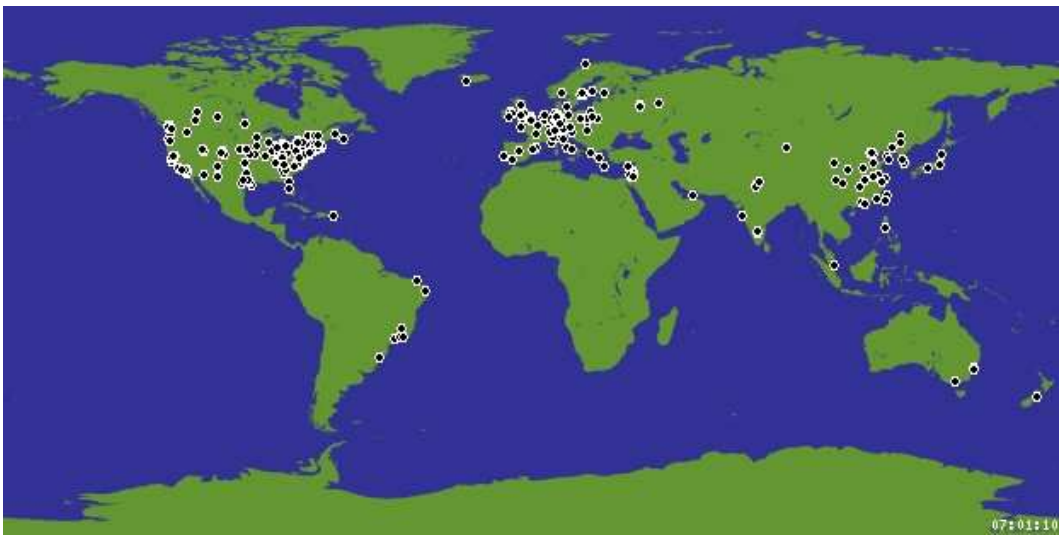


Figure 5.4: Current (27-06-2006) distribution of nodes in PlanetLab

The most important advantage in using PlanetLab is that experiments can be conducted under real-world conditions, and at a large scale. Applications using PlanetLab are widely distributed over the Internet, across multiple administrative boundaries.

Our application used PlanetLab machines to build 4 trees of different sizes. We evaluated our system on each of the trees separately. The characteristics of constructed trees are summarized in Table 5.1.

The hop-latency is the latency between pairs of nodes in the tree (between a child and its parent). The path-latency is the latency between the node in the tree and the root node. For both, we present their average and maximum value as well as the standard deviation (denoted by σ). All the latency values are in milliseconds.

Tracking these values will allow us to examine the influence of the latencies in the underlying TCP/IP network on the efficiency of communication in the P2P overlay network. Furthermore, as stated in Sections 4.2 and 4.3, the quality of information is expected to be

nodes	sites	nodes per site	hop-latency avg	hop-latency max	hop-latency σ	path-latency avg	path-latency max	path-latency σ	depth avg	depth max
10	8	1.3	78	172	59	78	172	59	1.0	1
40	30	1.3	91	242	59	104	324	67	1.1	2
80	61	1.3	73	304	67	120	434	81	1.4	3
160	103	1.6	88	348	78	148	553	98	1.7	2

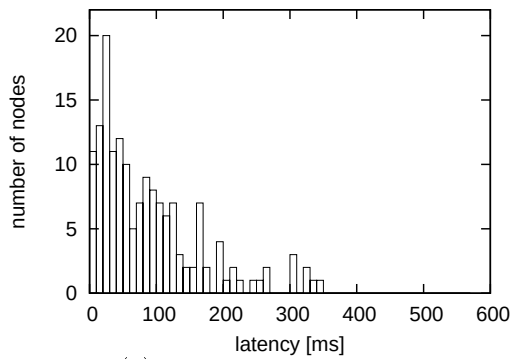
Table 5.1: Sets of nodes used in tests

directly connected to the average travel time of messages through the network in case of the push approach, and maximum travel time in case of the pull approach.

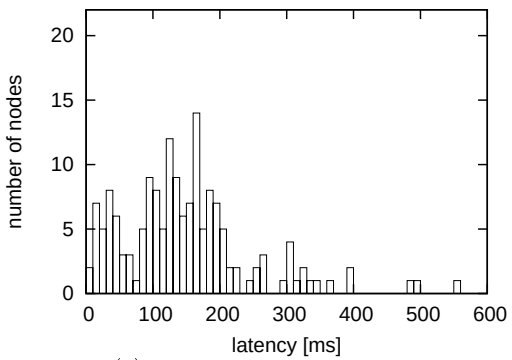
Nodes for the experiments have been chosen so that the number of sites was as high as possible. We managed to achieve the nodes per site factor as low as 1.6 for the tree of 160 nodes. This ratio for all nodes in PlanetLab is 2.1. We tried to minimize the number of child-parent TCP/IP connections within the same local area, because we wanted to build a large-scale system being able to work under many different conditions. Using many LAN connections inside trees would strongly influence the results.

For the visualization of nodes distribution for our tree of 160 nodes see Figure 5.5(a).

(a) visualization of nodes distribution



(b) hop-latency histogram



(c) path-latency histogram

Figure 5.5: Tree with 160 nodes

Characteristics of the largest tree will now be described in detail. Figures 5.5(b) and 5.5(c) present the distribution of hop and path-latency for our 160-nodes tree. Latency values are grouped by 10 ms.

Hop-latency values are mostly low, 90% of measurements fall below 200 ms, whereas 90% of path-latency values fall below 300 ms. Only 10 paths (i.e. 6%) have latency below 20 ms. The path-latency distribution within the range of 0-200 ms is balanced.

5.2.3. Assuring reproducibility

Testing applications in such an unstable environment as PlanetLab is not an easy task. In order to improve reproducibility of our tests, we introduced several additions to our system.

Generators

Our prototype implementation uses artificial generators as a source of measurements. In this way, we gain full control over the frequency and value of the measurement. Still, they mimic real sensors.

The generator is parameterized by the average interval between reporting new values to the system (I). The real interval is drawn with the Gaussian distribution with the mean value of I. The standard deviation is adjusted in such a way that approx. 99.7% of drawn values fall into the range of 50% to 150% of I. For outliers, the range boundaries are used.

The generated measurement values, in turn, are always in the range $\langle 0, 10000 \rangle$. They are randomly drawn in a similar manner to the interval. The previous value is always the mean for generating the new one. Again, the standard deviation is chosen in such a way that approx. 99.7% of the distribution falls into the range of 1250 from the mean value. The initial value is chosen randomly with Gaussian distribution with the mean value of 5000.

The parameters values are chosen so that the generator would mimic the behavior of the CPU load sensor.

Fixed nodes id

Pastry assigns ids to the nodes randomly. Consequently, each time the node joins the network, it most probably will receive a different id. Since Scribe multicast tree structure strongly depends on these ids, we decided to generate them differently. In our implementation, node id is computed as a SHA hash of the node's hostname.

As a result, each run of the application on the same set of hosts should result in the same tree built by Scribe (for very large sets, however, the tree structure may depend on the order the nodes contact the bootstrap group).

Root node

Our system, built on top of Pastry and Scribe, is able to deal with temporarily limited node responsiveness, which can occur due to increased CPU load or decreased network bandwith. The root node of the multicast tree, however, should remain stable throughout the tests. Otherwise, significant variance of results would be observed.

Scribe generates an id for the group based on the name provided by the application. To fix the root node (the rendez-vous point for the group), we chose the same basis for generating both the group and the root node id, namely the hostname. For our tests, we have chosen `planetlab1.cs.vu.nl` to be the root node.

5.2.4. Quality of information

Because of network latencies and software overhead, the answer to a query may become outdated, especially when the generator interval is low. The ability to compare the quality of information served by different approaches becomes a crucial issue. Therefore we define *staleness of information* as the time passed since the acquired information has stopped being correct. The precise definition will follow.

As stated in Section 5.2.1, our system implements only 'minimum' aggregation. Consequently, we provide all the definitions in the context of minimum values and queries. Additionally, we assume that we have a set of nodes, each generating values at random points in time.

Real minimum at the given point of time t is the minimum of values from all nodes at time t. The real minimum consists of the value and the list of nodes reporting this value.

Observed minimum is the value reported by the application as the minimum value for the network. The querying node records this value together with the index of the node the value originates from and the time it was generated by that node.

Staleness of an observed minimum at a given time t_s is equal to 0 only if the real minimum at t_s is equal to the observed value and the originating node of the observed value is in the list of nodes for the real minimum. If the observed minimum was correct at the time it was observed, but was not accurate at t_s, the staleness is the difference between the given time t_s and the time of the first real minimum after the observed. Finally, if the observed value has never been accurate, the staleness is the difference between t_s and the time the value was observed.

For the example calculations of the staleness see Figure 5.6.

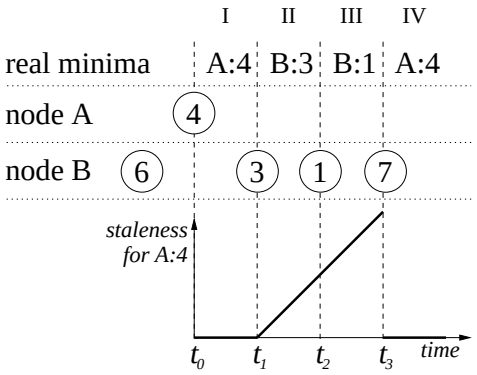


Figure 5.6: Staleness

In this example we have two nodes (A and B) generating new values at random points in time. The figure reveals also real minima for every section, e.g. *A:4* for I, meaning *value of 4 at node A*. For this example we assume that the observed value is *A:4* and for this value we calculate and plot the staleness.

In section I, the observed minimum is correct, i.e. the value (4) is equal to real minimum and the node (A) is in the list of nodes of real minimum. Thus, the staleness is equal 0 for $t \in \langle t_0, t_1 \rangle$.

At t_1 the new real minimum is present. Thus, the observed minimum *A:4* is no longer accurate. Since t_1 is the time of the first real minimum after the observed minimum, the staleness for $t \in \langle t_1, t_2 \rangle$ is $t - t_1$.

In section III, the observed value is still inaccurate. Although the real minimum changed

(to *B:1*), the time of the first real minimum after the observed minium remained the same. Thus, the staleness for $t \in \langle t_2, t_3 \rangle$ is $t - t_1$.

Finally, at t_3 node B generates the new value of 7. Thus, the global minimum changes again to *A:4* and the observed value becomes accurate once more. Thus, staleness for $t > t_3$ is 0.

5.2.5. Time

Most of the PlanetLab nodes synchronize time with time servers. Unfortunately, many nodes in the network do not synchronize at all or show significant inaccuracy in their clocks. In order to calculate the staleness of information and thus be able to compare different results, time differences between clocks in the network have to be computed.

In our system, the root node is responsible for analyzing data from all the other nodes. Consequently, the root node has to compute time differences between its clock and clocks of other nodes. To achieve that, we perform a very simple exchange of timestamps between nodes – see Figure 5.7.

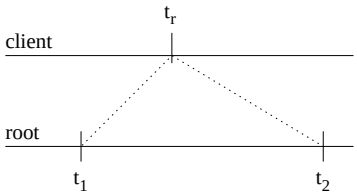


Figure 5.7: Computing time difference

In the figure, the root node contacts one of other nodes (client). After establishing the connection, the root node sends a request to the client, storing the time of this communication (t_1). The client returns it current time – t_r. The root node timestamps this information upon receiving (t_2).

Based on three collected values (t_1, t_r, t_2), the root node is able to estimate the latency (L) with the given node and, most importantly, the time difference (Δ_t). They are computed as follows:

$$\begin{aligned} L &= t_2 - t_1, \\ \Delta_t &= t_1 - (t_r - L/2)\,. \end{aligned}$$

The latency is estimated as the round-trip time of a very small TCP-packet, containing only timestamps. The time difference, in turn, is computed as a difference between root clock value in t_1 and estimated client clock value at the same point of time.

This approach is based on the assumption that asymmetry of links between nodes is negligible and one-way trip time can be estimated by the half of the round-trip time.

5.2.6. Test methodology

Building each of the four trees used in our tests required a few steps. First of all, our application had to be started on the set of nodes. Then, Pastry initialized all its required connections and Scribe constructed the multicast tree. At that moment, the tree was set up. Next, the root node investigated the tree structure, collecting information about depth and latencies. Finally, it could contact each of the nodes in the tree to compute time differences.

The entire starting process was taking as much as 20-30 minutes for the tree of 160 nodes. The Pastry phase was taking only a couple of minutes. Then, Scribe needed up to 15 minutes to stabilize the tree, since it periodically checks for the best known parent and needs to exchange a few messages to optimize the tree structure. Finally, the application needed 5 to 10 minutes for calculating the latencies and time differences.

For each tree a series of tests was conducted. We tested performance of our system for each approach (pull, push, and mixed) and several values of generators interval (400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 3000, 4000, and 5000). Every single test was programmed for one minute and was repeated three times, from which the average performance was computed. The results are presented in the next sections.

5.3. Pull approach

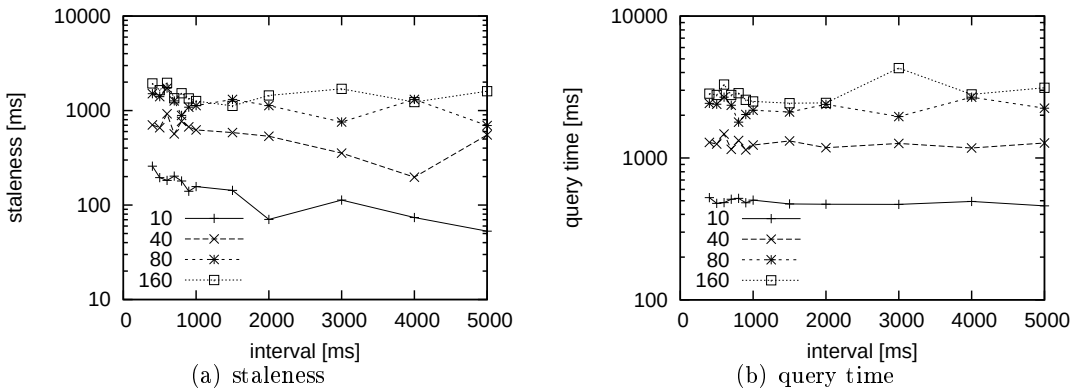
In the pull approach no information cache is maintained. Every query has to be answered by collecting current measurement values from the entire tree. For detailed description of the architecture see Section 4.2.

The tests were conducted as follows. For each tree size and each interval three runs of tests were executed. In each run the root node was submitting queries asking for the minimum value in the tree. A single run consisted of 30 queries submitted at the rate of 1 query per 2 seconds. The interval between queries was chosen high in order not to introduce too much overhead to the system and not to influence the performance. This decision was motivated by the observation that the pull approach is most suitable for systems with relatively low query rate.

For each answer returned by the system its staleness was computed. Additionally, we tracked time the system needed to answer the query.

5.3.1. Staleness and query time

Figure 5.8 presents the overview of the results. The graphs show the average staleness and the average propagation time (log-scaled Y-axes) in relation to generator intervals for different tree sizes.



(a) staleness (b) query time

Figure 5.8: Staleness and query time vs. interval in pull approach

We observe significant differences in the quality of information between trees of 10 and 40, as well as between 40 and 80 – see Figure 5.8(a). However, the difference between 80 and

160 is noticeable smaller. This can be explained by the fact that the difference in maximum path-latency acts similarly.

Furthermore, we observe that the average staleness tends to decrease with the increase of the interval. It also seems to be relatively stable for larger (above 1000 ms) intervals. We present detailed values of average staleness in Table 5.2.

tree size	staleness for intervals 400-900	1000-5000
10	193	102
40	716	474
80	1314	1056
160	1627	1394

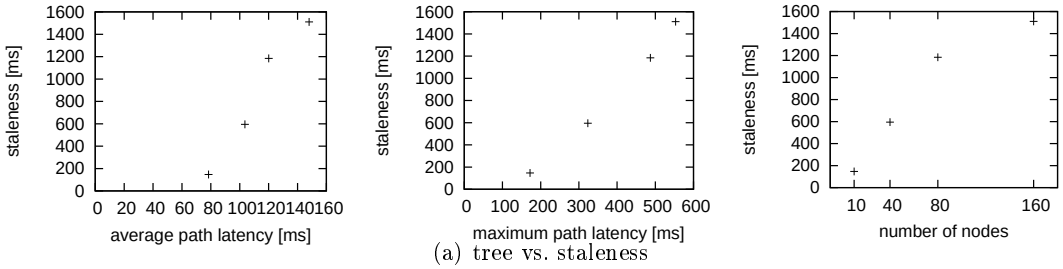
Table 5.2: The average staleness for pull approach

The results suggest that the larger our system grows the worse is the quality of service it provides and the less sensitive to generator intervals it is.

Figure 5.8(b) depicts the query time. It confirms the intuition that the query time does not depend on the interval. Furthermore, it shows that the query time increases with the size of the tree. This relation will be investigated in detail in the following section.

5.3.2. Tree size

The graphs in Figure 5.9 show the relation between the tree size (expressed in the number of nodes and the maximum path-latency) and both the query time and the staleness. Figure 5.9(a) shows the staleness averaged over all intervals for different tree sizes. The graphs in Figure 5.9(b) show the query time in milliseconds (averaged over all generator intervals) for different tree sizes.



(a) tree vs. staleness

(b) tree vs. query time

Figure 5.9: Tree characteristics vs. query time in pull approach

We can observe that our expectation expressed in Section 4.2 is plausible. Indeed, the quality of information (measured with the staleness) is linearly proportional to the maximum

path-latency in the tree. The query time reveals the same relation to the maximum path-latency. Multiplying the maximum path-latency by the factor of 2 most probably results in doubling the query time and the staleness.

We can also note that the number of nodes itself has rather indirect influence on the query time and the staleness. The relation does not seem to be linearly proportional.

5.4. Push approach

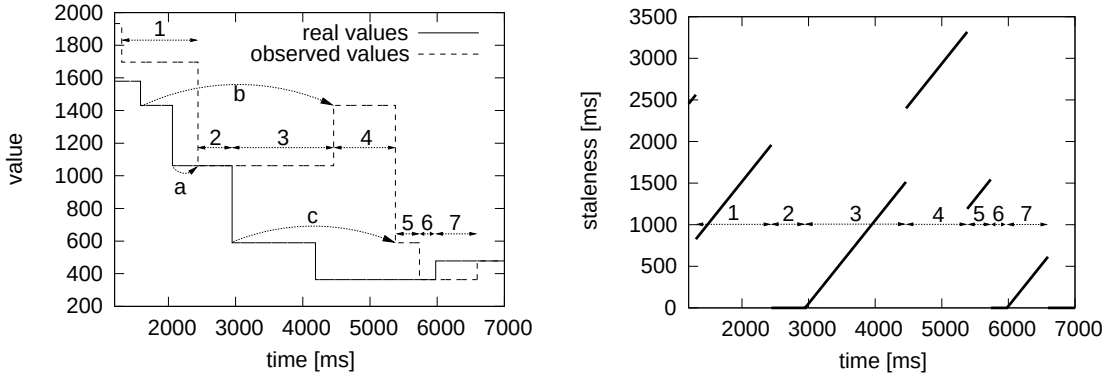
Nodes in the push approach cache information about the minimum value in the subtrees represented by their children. Therefore, the minimum queries can be answered immediately. For the detailed description of this approach see Section 4.3.

The tests of this approach were conducted by running the system for 60 seconds, 3 times for each tree size and interval. During each run, the nodes were constantly exchanging information in order to update their caches. For the test period, we measured the staleness of information in the root node at every millisecond. The question we wanted to answer was 'what quality of information can be expected by users submitting queries at random points in time'.

Additionally, we tried to investigate the overhead that our system introduces by measuring the number of messages per second. We also measured the average propagation time of information through the tree. We start, however, with investigating the delivery of events to the root node.

5.4.1. Event delivery

Figure 5.10 presents the fragment of one of the test runs on the tree of 160 nodes for the 1500 ms generator interval.



(a) real and observed values

(b) staleness

Figure 5.10: Values and their staleness in push approach

Figure 5.10(a) visualizes the real values (global minima) and the values reported by the system to the root node (observed values). Arcs with numbers mark the most interesting errors the system makes. Figure 5.10(b) presents staleness of information for the same period. In both figures, lines with arrows split the period into several periods.

In the first period the system reports an incorrect value to the user. It has its reflection in the staleness of this information. Since this information was never correct, its staleness in the beginning of period 1 is non-zero. Naturally, it grows throughout the period.

Arc a denotes a simple delay in the event delivery to the root node. The real minimum needed approximately 200 ms to be delivered to the root node and to be considered an observed value. Thus, in period 2, the staleness equals 0.

At the beginning of period 3, the real value changed. The system, however, still served the same value. Again, throughout the entire period the staleness grows with each millisecond starting with 0.

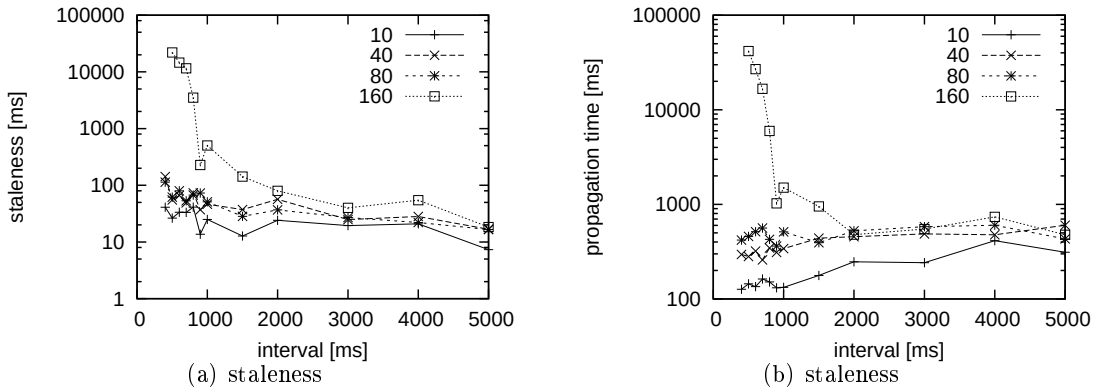
Arc b marks an interesting event reordering. The value of 1400 was considered observed minimum after the value of 1050. The real values came in reversed order. Consequently, the observed minimum of 1400 was stale from the very beginning, i.e. from the moment it was considered as the answer.

Period 5 begins with a late delivery of the value of 600, denoted by arc c. The system started to serve this as an observed value when it stopped being the real minimum. Consequently, its staleness is non zero throughout the entire period.

Period 6 is the time of system serving accurate information, thus staleness is 0. However, the next period is the time needed by the root node to be informed about a change in the real minimum. Throughout this period, the system serves an inaccurate information. Thus, the staleness grows.

5.4.2. Staleness and propagation time

The staleness and propagation time for each interval and tree size is depicted in Figure 5.11. Y-axes for the staleness and the propagation time are log-scaled.



(a) staleness (b) staleness

Figure 5.11: Staleness and propagation time vs. interval in push approach

In Figure 5.11(a), we observe relatively low differences in staleness between different tree sizes for intervals above 1000 ms. For lower intervals, however, the staleness tends to increase dramatically with the decrease of the interval. We present detailed values of average staleness in Table 5.3.

tree size	staleness for intervals 400-900	1000-5000
10	31	18
40	70	35
80	75	64
160	10389	140

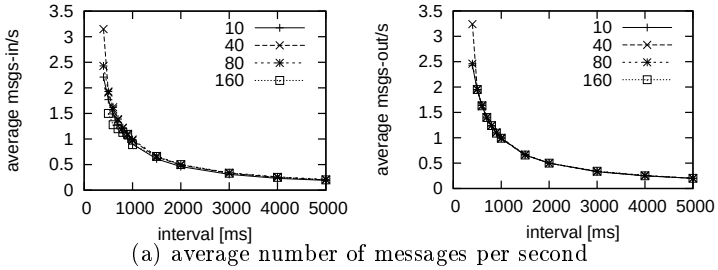
Table 5.3: The average staleness for push approach

The staleness for generator interval of 500 ms for our largest tree was as high as 22 seconds. This extreme value was caused by the tremendous amount of events in the tree. The system became overloaded by all the messages it had to process. Our system was not able to handle the overwhelming amount of events for the interval of 400 ms. Therefore, we do not present results for this interval for the largest tree.

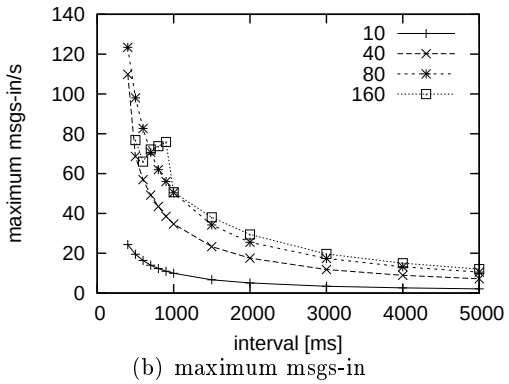
Figure 5.11(b) reveals the extreme propagation time for events in the tree of 160-nodes. For all other trees, this value is relatively stable independently of the interval. For the largest tree, however, it grows extremely, reaching the value of 42 seconds for the interval of 500 ms.

5.4.3. Communication overhead

In order to thoroughly investigate the impact of the communication overhead on the system, we present the number of messages per second rate for the root node and average for the entire tree. The results are depicted in Figure 5.12.



(a) average number of messages per second



(b) maximum msgs-in

Figure 5.12: Messages per second in push approach

Figure 5.12(a) presents the average number of messages per second. The average is computed over all nodes in the network and presented in two graphs: incoming and outgoing messages separately. Interestingly, this number depends only on the interval, not on the size of the tree. This can be explained by the fact that adding a child to the tree increases messages per second rate at its parent and grandparents, but contributes a value of 0 to the average at the same time.

The graphs reveal that the number of messages is inversely proportional to the interval. Increasing the interval by the factor of 2, decreases the number of messages per second by the same factor.

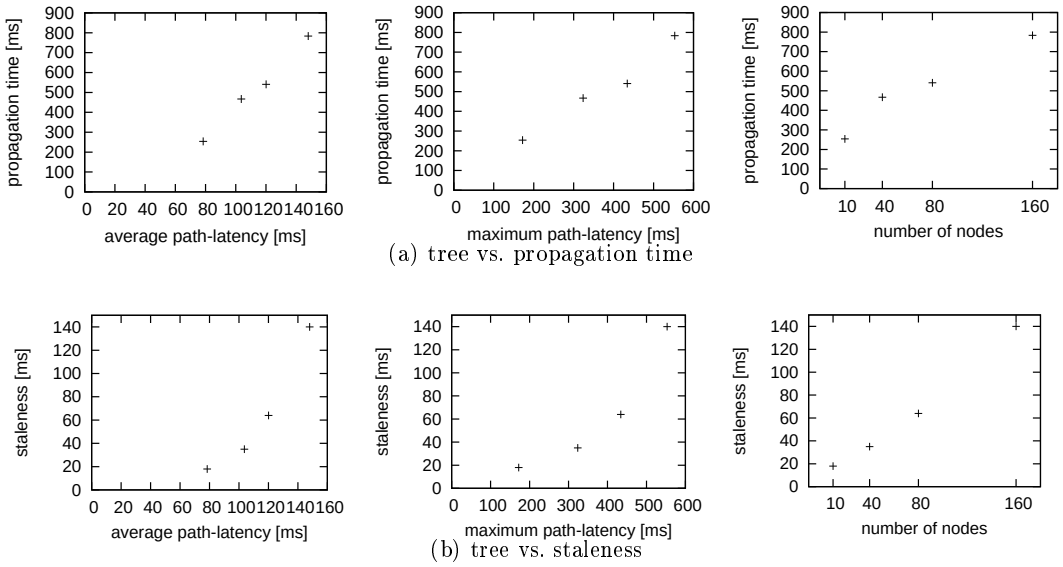
The more interesting observation come from Figure 5.12(b). First of all, we observe that the maximum number of incoming messages per second grows with the tree size. Secondly,

for each tree size the interval is inversely proportional to the maximum number of messages, with exception of the largest tree.

As we can see, the relation between the interval and the message rate for the tree of 160 nodes is not proportional below the interval of 1000 ms. For intervals in the range 500-1000 ms it behaves very strangely. Most probably this is caused by too large communication overhead, as a relatively steep increase in message rate between intervals of 1000 and 900 ms may suggest. As we could see in Figure 5.11(b), this resulted in a significant delays in messages delivery.

5.4.4. Tree size

The final evaluation of the push approach is investigating the relation between the tree size and both the propagation time and the staleness. The results are presented in Figure 5.13. The staleness and the propagation time is averaged over intervals within the range 1000-5000 ms, as those could be considered as representative for the stable system.



(a) tree vs. propagation time

(b) tree vs. staleness

Figure 5.13: Tree characteristics vs. propagation time in push approach

The relation between path-latency (both average and maximum) seems to be nearly linearly proportional. As in the previous approach, this relation is weaker in case of the tree size.

The staleness shows exponential relation with the tree size expressed in average and maximum path-latency. Acquired data, however, do not provide us with sufficient information to support or reject the expectation from Section 4.3 that the quality of information is proportional to the *average* and not *maximum* path-latency.

Surprisingly, the relation between the number of nodes and the staleness seems to be nearly linear.

5.5. Mixed approach

The mixed approach merges concepts of the push and pull models. Leaf nodes work in the push mode, the others remain in the pull model. For detailed description of this approach

see Section 4.4.

Tests of this model were conducted similarly to the pull approach. The root node was submitting minimum queries at a fixed interval of 2 seconds. The answers were recorded, together with their staleness and time the system needed to answer them.

In this series of tests, however, we skip the tree of 10 nodes. Since its depth was 1, the mixed approach in this case was reduced to the push mode. Simply, each node in such a tree is a leaf node of the root node. Consequently, the push area spans over the entire tree.

5.5.1. Push and pull zones

The crucial observation for tests are sizes of the pull and push areas in our trees. Their characteristics are shown in Tables 5.4 and 5.5.

tree nodes	push nodes		hop-latency avg	max	σ
40	35	(88%)	93	242	65
80	66	(80%)	63	304	63
160	146	(91%)	87	348	79

Table 5.4: Push area in mixed approach

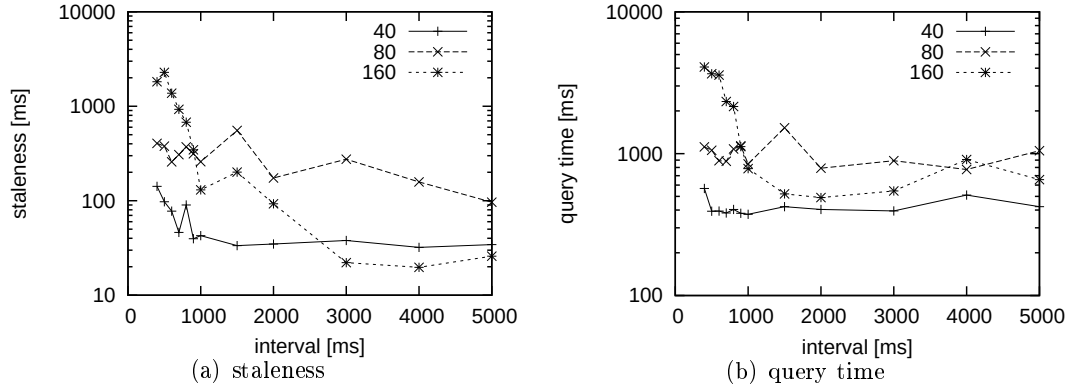
tree size	pull nodes		path-latency avg	max	σ	depth avg	max
40	5	(12%)	88	156	65	1.0	1
80	16	(20%)	131	267	58	1.1	2
160	14	(9%)	108	302	72	1.0	1

Table 5.5: Pull area in mixed approach

An interesting observation is that the largest (in terms of the average depth, the number of nodes, and the average path-latency) pull area belongs to the middle-sized tree.

5.5.2. Staleness and query time

Figure 5.14 summarizes the tests results. It presents the staleness and the query time on the log-scaled Y-axes in relation to generator intervals.



(a) staleness (b) query time

Figure 5.14: Staleness and query time vs. interval in mixed approach

Again, as in the previous approach, both the staleness and the propagation time seem to be relatively stable across different intervals. The only exception is our largest tree. For intervals from the range 400-1000 ms the staleness grows rapidly to the value of 1.8 seconds for the interval of 400 ms. The similar rapid growth can be observed for the query time, reaching the value of 4.1 seconds for the same interval.

We present detailed values of average staleness in Table 5.6.

tree size	staleness for intervals 400-900	1000-5000
40	82	36
80	339	253
160	1238	82

Table 5.6: The average staleness for mixed approach

As in the push approach, there are some differences between low and high generator intervals. They are significant for the largest tree size.

We can also observe in Figure 5.14(a) that the tree of size 80 behaves surprisingly bad and the largest tree acts surprisingly good for the intervals greater than 1000 ms. The possible explanation for that is the size of the push area.

As the previous sections showed, the pull approach seems to give more balanced results for different intervals, whereas the push approach tends to show significant differences in the quality of information between lower and higher intervals.

The pull area for the tree of 80 nodes (20%) is remarkably larger than for the tree of 160 nodes (9%). This might be the reason for worse yet more stable results for the 80-nodes tree.

On the other hand, the wide push zone in the largest tree is the rapid decrease in the quality of information for lower intervals (below 1000 ms), typical for the push approach.

5.5.3. Communication overhead

As in the push approach, the maximum incoming messages per second rate acts strangely for low intervals and large trees. Figure 5.15 presents the relation between the message rate and intervals for different trees.

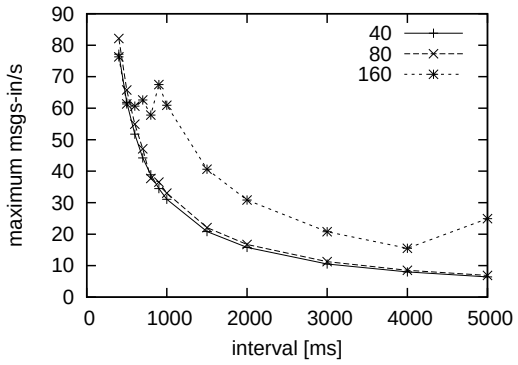


Figure 5.15: Messages per second in mixed approach

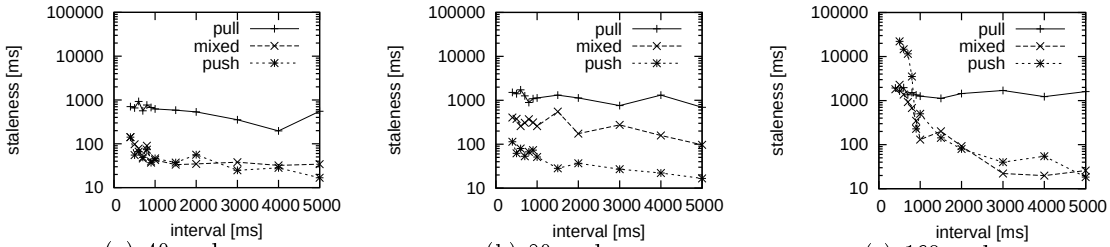
As we can see, the maximum messages per second rate is significantly larger for the 160-nodes tree. As in the push approach, the system probably becomes flooded by the number of messages it has to process and consequently, the quality of service decreases.

5.6. Comparison

In this section, we present the comparison of different approaches at the per-tree basis. We do not present the tree of 10 nodes, since the mixed approach was not evaluated for that tree.

5.6.1. Quality of information

At first, we look at the quality of information for different tree sizes and intervals. Its values are visualized in Figure 5.16.



(a) 40 nodes (b) 80 nodes (c) 160 nodes

Figure 5.16: Staleness in three approaches

For the 40-nodes tree (see Figure 5.16(a)), we observe that results of the mixed and push approaches are very close to each other. This was caused by the fact that in that case the tree in the mixed approach was practically reduced to the push approach, since the push area covered 88% of the nodes.

The results in Figure 5.16(b) are closest to our expectations. The quality of information for the mixed approach is clearly worse than in the push approach and significantly better than in the pull model.

Figure 5.16(c) presents the problems both push and mixed approaches had with low generator intervals. The staleness grows massively for intervals within the range 400-1000 ms. We can also observe that in this case, as in the case of 40 nodes, the push and mixed approaches give similar results. The push area in the mixed approach was as large as 91%, which might have been the cause.

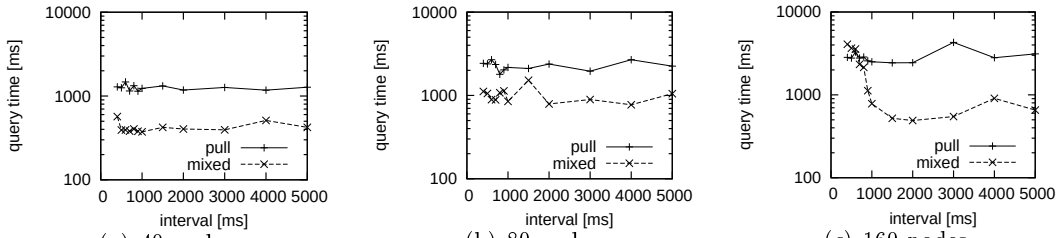
The largest tree shows an interesting feature of the mixed approach. It allowed the system not to get overflooded by communication in lower intervals. The mixed approach returned similar results to the pull approach and the quality of information did not decrease as extremely as in the push approach. For intervals above 1000 ms, in turn, the mixed approach was as good as the push approach.

5.6.2. Query time

The query time for different trees is depicted in Figure 5.17. Naturally, its value for the push approach is 0, since the system is always capable of returning the immediate answer.

The query time in the mixed approach for all trees is significantly smaller than in the pull model. Moreover, the query time depends strongly on the size and the characteristics of the pull area, which is only a fraction of the entire tree in the case of the mixed model.

Figure 5.17(c) confirms that the mixed approach behaves oddly when the interval is smaller. It inherits the weakness of the push approach and is not able to handle huge event traffic efficiently.



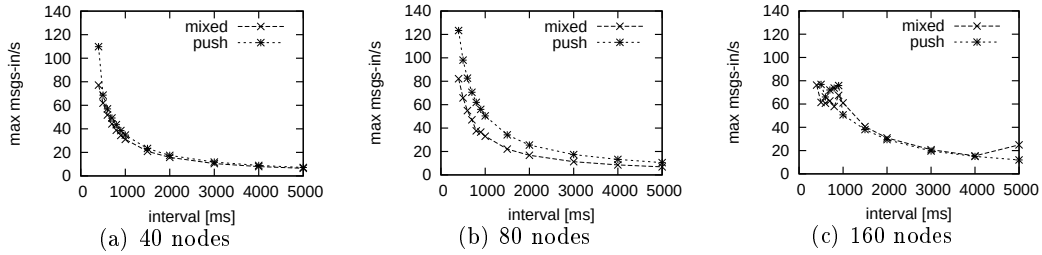
(a) 40 nodes (b) 80 nodes (c) 160 nodes

Figure 5.17: Query time in three approaches

Figures 5.16 and 5.17 together show that the staleness and the query time are bound. Decreasing the query time (by using a different approach) results in a better quality of information.

5.6.3. Communication overhead

To see the trade-off between the quality of information and the communication overhead, we will now compare the maximum incoming messages per second rate for the mixed and push approaches – see Figure 5.18. Naturally, there is no communication overhead for the pull approach.



(a) 40 nodes (b) 80 nodes (c) 160 nodes

Figure 5.18: Maximum messages per second in three approaches

The tree of 80 nodes shows that the smaller the interval is, the larger the difference between messages rates for the mixed and pull approaches. The value of 123 messages per second for the push approach at the interval of 400 ms is significant. It results in a message being delivered to the root node every 8 ms. Processing capabilities of the node might be a serious limitation.

Nevertheless, the relation between maximum messages per second rate and the interval seems to be accurate – it is inversely proportional, as for the 40-nodes tree in Figure 5.18(a) and the mixed approach in Figure 5.18(b). It suggests, that although the system was seriously loaded, it was still capable of processing all the messages on time. Consequently, we do not observe any staleness breakdown for low intervals in the case of 80 nodes as we see for 160 nodes. It might suggest that implementing a method of dealing with the overwhelming network traffic (e.g. dropping some messages) could make the system resilient to large tree sizes and low intervals without loosing too much of the quality of information.

Since our system does not control the communication amount, we observe a breakdown of the maximum messages per second rate for low intervals for the 160-nodes tree – see Figure 5.18(c). This behavior was already discussed in Section 4.3. Most probably it was caused by the overwhelming communication overhead. The TCP/IP queues became overloaded and messages were significantly delayed.

Chapter 6

Conclusions and future work

6.1. Conclusions

This thesis presents the architecture and evaluation of a peer-to-peer Grid monitoring system. The concept of P2P network has been incorporated into the system in order to assure scalability, robustness, and manageability.

The architecture proposal has been preceded by a thorough study of existing Grid monitoring systems and other relevant systems. As a result, we have identified and classified possible types of queries a Grid user may want to submit. We have proposed two categories of queries: direct measurements and resource discovery. The latter encompasses advanced queries with resource selection and rankings. We have also concluded that a decent Grid monitoring system should concentrate on the efficient and convenient support of advanced resource discovery queries and regard direct measurements as a supplementary information.

Furthermore, we have classified types of information Grid monitoring system has to process in order to answer user queries. We have identified node measurements, job queues, and network measurements.

The proposed system architecture emphasizes resource discovery queries. It uses P2P to deal with the environment instability, as well as to assure scalability of the system. Additionally, P2P overlays the network with an acyclic graph, which our system exploits. We have proposed three approaches to answering queries: push, pull, and mixed. The push approach maintains a cache, thereby allowing for immediate answers. The pull model does not require additional communication and answers queries by collecting information from the entire tree. The mixed approach joins the two concepts by splitting the network into two collaborating areas: the push zone and the pull zone.

In order to evaluate the architecture, we have introduced the definition of staleness as an indicator of the quality of information returned by the Grid monitoring system. We have prototyped the architecture to test the proposed approaches. The implementation uses Scribe – the application level multicast protocol. We have established a testbed using PlanetLab resources and tested our system on the networks of 10, 40, 80, and 160 nodes. During the experiments, we have tried to relate the quality of information to the tree size and the interval, at which new measurements are delivered to the system.

The pull approach has shown a noticeable trade-off between the staleness and the interval. Additionally, this approach has revealed a linearly proportional relation between the staleness and the maximum path-latency in the tree.

The push model, in turn, has shown that the staleness depends on the interval. For lower values of the interval, the staleness grows significantly. For larger values, it behaves relatively

stable. Additionally, we have noticed that the communication overhead becomes an important problem for low intervals and large tree sizes.

Finally, the efficiency and the quality of information in the mixed approach strongly depend on the size of pull and push zones. The larger the pull area is, the more stable and less accurate the results are, the less communication overhead is introduced, and the longer the query time is.

Our tests have confirmed the intuition that the immediate answers and the high quality of information come at the price of communication overhead.

We conclude that the mixed approach is the most promising one. A system of desired quality of information and the communication overhead can be built by appropriately splitting the network into push and pull zones. Therefore, the system can be easily customized to meet the specific requirements of the given environment.

6.2. Future work

For future work we propose two paths: further evaluation and/or implementation.

As for evaluation, simplifications made by our work can be relaxed and tested. In particular, relaxing the assumption that only the root node can submit queries should be further investigated. Additionally, testing the efficiency of tree pruning for queries with constraints might give interesting results.

Extending the prototype into the complete application would require designing sensor modules, prediction library, and implementing all types of queries and aggregation.

However, most interesting and tempting direction is considering the self-adaptive Grid monitoring system. Such a system could have two levels of adaptation: queries and the environment.

The self-adaptive Grid monitoring system might choose the query answering strategy (push, pull, or mixed) accordingly to the current query load. Implementing this behavior in the distributed manner seems to be challenging.

The environmental self-adaptation, in turn, would involve the network deciding which strategy to use (push or pull). The decision would be made locally (by single nodes or group of nodes) and be based on e.g. the quality of network connections with other nodes. The node would then switch from the push to the pull approach when it discovers that the latency to a given node has significantly decreased.

Appendix A

Software archive

This appendix describes the content of the software archive being an attachment to this thesis and gives instructions on how to compile and run the application and how to perform tests.

Archive

The content of `p2pgms.tgz` is as follows:

1. `compile` – a script to compile the source code and build the jar file.
2. `freepastry.params` – a configuration file for the Pastry.
3. `MANIFEST.MF` – a manifest file defining 'main-class' of the project.
4. `p2pgms.jar` – the compiled and ready to run project.
5. `pastry.jar` – Pastry library (ver. 1.4.4) required to run the project.
6. `run` – a script for running the project.
7. `src/` – the directory with the source code of the project. The class containing the `main` method is `d.Main`, the most important Scribe class is `d.MyScribeClient`. The remaining classes are organized in the following packages:
   - `d.cache` – classes responsible for maintaining the cache in the push approach,
   - `d.msg` – all the message classes in the project,
   - `d.pullQueries` – classes responsible for query processing in the pull and mixed approaches,
   - `d.tools` – classes for latency and time differences computations,
   - `tests` – generator as well as tests logic for the root node.

Compiling

The project requires `pastry.jar` for compilation. The script `compile` invokes the Java compiler with the proper classpath. Additionally, it builds the project jar file (`p2pgms.jar`).

Running

For starting the application, `run` can be used. It simply invokes `java -jar p2pgms.jar`.

The application retrieves the bootstrap group from the DNS. It uses `pastry.mejdys.globule.org` domain name, which can be configured in the attribute `DNS_NAME` of the class `d.Config`. The application tries to contact any of the hosts in the retrieved list. If none can be contacted, and the host determines it belongs to the bootstrap group (its name is simple in the list), a new Pastry ring is started.

It is important to start the nodes in the proper order, i.e. to first start one of the bootstrap nodes. Then, preferably, the rest of them should be started, and finally, the remaining nodes. This will assure efficient network joining procedure.

Each node, after the successfull start, allows for giving commands to it. Only the root node's console should be used. It accepts the following commands:

- `m` – collects time differences with all nodes in the tree,
- `s` – starts tests in the push mode,
- `l` – starts tests in the pull mode,
- `p` – starts tests in the mixed mode,
- `t` – collects information about the tree,
- `?` – prints help.

The testing scenario is as follows:

1. Start all the nodes and wait till the tree stabilizes. Mind, that since the application was prepared to the tests only, it does not handle the changes in the tree structure properly. Thus, during the early stage of initialization, when lots of nodes are joining the network, some commands may fail.
2. Collect the tree information (option `t`).
3. Collect the time differences information (`m`).
4. Perform tests. This step can be repeated many times.
5. Finish the application by killing it at each node.

Bibliography

[1] Brisco, T. DNS Support for Load Balancing. RFC 1794 (Informational), Apr. 1995.

[2] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. Design of a scalable event notification service: Interface and architecture. Tech. Rep. CU-CS-863-98, Department of Computer Science, University of Colorado at Boulder, September 1998.

[3] Castro, M., Druschel, P., Kermarrec, A.-M., and Rowstron, A. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC) 20*, 8 (Oct. 2002), 100–110.

[4] Drost, N., van Nieuwpoort, R. V., and Bal, H. E. Simple locality-aware co-allocation in peer-to-peer supercomputing. Accepted for publication at GP2P: Sixth International Workshop on Global and Peer-2-Peer Computing, May 2006.

[5] Foster, I., and Iamnitchi, A. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proceedings of the 2nd International Workshop on Peer-toPeer Systems (IPTPS'03)* (February 2003), pp. 118–128.

[6] Hoschek, W. Peer-to-peer grid databases for web service discovery. *Concurrency: Practice and Experience 00* (2002), 1–7.

[7] Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., and StevenLim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials 7*, 2 (April 2005), 22–73.

[8] Maassen, J., van Nieuwpoort, R. V., Kielmann, T., Verstoep, K., and den Burger, M. Middleware Adaptation with the Delphoi Service. Accepted for publication in Concurrency and Computation: Practice & Experience, 2005.

[9] Massie, M. L., Chun, B. N., and Culler, D. E. The Ganglia Distributed Monitoring System: Design, Implementation And Experience. *Parallel Computing 30*, 7 (2004).

[10] Pallickara, S., and Fox, G. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Middleware* (June 2003), vol. 2672 of *Lecture Notes in Computer Science*, Springer, pp. 41–61.

[11] Park, K., Pai, V. S., Park, K., and Pai, V. S. CoMon: A mostly-scalable monitoring system for PlanetLab. *SIGOPS Operating Systems Review 40*, 1 (2006), 65–74.

[12] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. A scalable content-addressable network. *SIGCOMM Computer Communication Review 31*, 4 (2001), 161–172.

[13] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware* (November 2001), vol. 2218 of *Lecture Notes in Computer Science*, Springer, pp. 329–350.

[14] Sacerdoti, F. D., Katz, M. J., Massie, M. L., and Culler, D. E. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of the IEEE Cluster 2003 Conference, Hong Kong* (2003), pp. 289–298.

[15] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference* (August 2001), pp. 149–160.

[16] Wolski, R., Spring, N., and Hayes, J. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems 15*, 5–6 (1999), 757–768.