Information Aggregation for Load Balancing in a Distributed System of Web (Grid) Services

A Framework of Aggregation Algorithms

M A R C   S C H N E I D E R

Master of Science Thesis
Stockholm, Sweden 2007

ICT/ECS-2007-60

Information Aggregation for Load Balancing in a Distributed System of Web (Grid) Services

A Framework of Aggregation Algorithms

Marc Schneider

Examiner
Vladimir Vlassov

Department of Electronic, Computer and Software Systems (ECS)
School of Information Technology and Communication Technology
Royal Institute of Technology KTH

Industrial Supervisor
Konstantin Popov

Swedish Institute of Computer Science
SICS

The Royal Institute of Technology
Stockholm, April 2007

Table of Contents

Table of Figures

Table of Tables

Table of Graphs

1 Abstract

Grid computing is the next logical step in distributed computing. The Grid allows us to share resources across administrative boundaries. The Open Grid Service Architecture OGSA defines Grid Services based on Web Service technology. The overall usage of Grid is expanding very fast and indulge large-scale systems spanning many organizational borders. To solve scalability issue of future Grid systems, technology as known from P2P systems can be engaged. Even though P2P and Grid systems have different origins, they tend to converge towards each other facing a common future in large scale resource sharing technology.

Through a background study, the thesis lightens up and discusses properties and architecture of P2P, Web Services and Grids. A survey of load balancing systems in the area of Distributed Systems will bring us into the context of real applications. The surveyed systems are discussed briefly and compared with each other.
I define a system model based on request routing in the problem space for large dynamic systems with the focuses on the European project Grid4All. I demonstrate that the model can be achieved in two components, where the first is gathering the information and the latter uses that information for load balancing.
I propose and evaluate practical algorithms for information aggregation in a structured P2P overlay. The aggregated information is the future input for load balancing algorithms. Actual load balancing is kept as a future work, where the contribution of this thesis goes to the aggregation of information algorithms.

2 Acknowledgements

I wish to thank my examiner Vladimir Vlassov and my industrial supervisor Konstantin Popov for their valuable comments, guidance and work we conducted together. I appreciated very much working close with them; they are leading researchers in the P2P field. Many thanks to the fantastic environment and people at SICS. I enjoyed a lot being part of this unique institute, which indulged my work in many aspects.
In the same go I would like to thank KTH which offers opportunities for international students participating a full MSc course. Its a “once in a life time opportunity” which has to be retained.
Special thanks goes to my parents Kathrin and Fritz, which supported me to make this education possible.

3 Introduction

Sharing resources among organizational and institutional boundaries needs an infrastructure to coordinate resources of boundaries within so called virtual organizations. Grid technology builds the infrastructure for virtual organization. Such infrastructure should offer a easy management of forming virtual organizations, sharing resources, discovering services and consuming services.

To solve these requirements, open and extensible standards must be employed. Thus allowing a broad interoperability and allowing the overall Grid technology being developed and evolved. Attractive technologies from Web Services are adapted: Discovering, look-up and invocation of services. In the recent years Web Services became the drive of Grid infrastructure.

Grid systems are becoming commercial and changing their face to a main-stream alike paradigm. Companies hiring out storage and computational power. Many research projects going on, which intend to compound computational power between research institutes. Grid systems are growing fast in the recent years. Centralized management of Grid systems are not scalable, and must be evolved using scalable technologies such as known from Peer-to-Peer systems.

Peer-to-Peer technologies are scalable and self managed. P2P system have the same idea of resource sharing as in Grids but have different views how resources are shared. Anyhow, P2P systems use overlay networks established on top of the existing network. The overlay is a self managed logical network which uses connectivity information from peers. These systems are highly scalable and their technology can be exploited to be used to make Grid systems scalable. Much research is going on to adapt P2P technology into Grids. In the recent years researches have noticed that the evolution of P2P systems and Grids converge to each other.

3.1 Goals and Expected Results

The goals of the thesis are the following

1. Background study of related work on Grids, Web Services and P2P
2. Survey of load balancing in the context of Distributed Systems
3. Experiment with the Globus Toolkit 4, by studding and deploying Grid Services together with Sotomayors book [1]. This give an insight of a real Grid System and hands on programming Grid Services.
4. Proposal and evaluation of algorithms, directing towards load balancing, using the structured overlay DKS developed at KTH and SICS. The main features of the Grid Service is great scalability and self management.

Expected Results

1. The survey of load balancing in a distributed system: show different technologies used to solve load balancing within Distributed Systems. We prefer to find applications where it allows to scale to a large number of members.
2. A system model for solving load balancing for a distributed system of Web (Grids) services: The model should be scalable and run within a dynamic environment.
3. Proposing algorithms for information aggregation for large-scale dynamic system.
4. Results of the evaluation of the algorithms.

We expected to build a prototype for load balancing. We have adapted the work to give a evaluation of the developed algorithms which covers the aggregation of information. Future work can be based on this thesis to propose and implement load balancing mechanisms.

4 Background study on P2P, Web Services & Grids

4.1 P2P Systems (overlay network)

P2P systems evolve fast and are an emergent technology in the future of the Internet, forming a new paradigm of computing and resource sharing. Actually, P2P form an overlay network on top of the Internet with some important properties relaying on its decentralized and non-hierarchical architecture such as self-organizing, massive scalable and robust in Internet sized networks.
The ACM Computer Survey [2] points out that there is not a general agreement on what is and what is not a peer-to-peer. In respect to sharing resources directly with other peers and be able to treat instability they define P2P as this:

> *Peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority.*

[2] points out that the definition is encompassing the different level of “peer-to-peeryness” (or decentralisation), ranging from a pure decentralized system to a partially centralized systems such as Napster [3].

The following characteristics are the main issues for developing and deploying P2P applications: *Decentralization, Scalability, Anonymity, Self-Organization, Cost of Ownership, Ad-Hoc, Connectivity, Performance, Security, Transparency and Usability, Fault Resilience, Interoperability*.

4.1.1 P2P overlay network structure

P2P overlay networks form a self-organizing system of peers. Overlays are logical networks, built on a physical communication network. They consist of 5 main layers depicted in Figure 1.

The first one is the *Network Communication layer* and describes how the network connectivity characteristics are (i.e. on a mobile device, desktop machine). Connectivity between peers is typically in a ad hoc manner.

The *Overlay Network Layer* manages the peers and is responsible for peer discovery, optimal routing and location look up.

The *Feature Management Layer* handles security, resource management and the resilience. Security deals mostly with some central authorities, albeit new distributed technology are developed against denial of service and reputation.

The *Service Specific Layer* (aka Class-Specific Layer) concerns itself with a bunch of classes enabling “features” on the infrastructure (Add-ons for supporting the P2P substrate). Such classes are scheduling (applies to compute intensive applications), meta-

data (applies to content and file management applications), messaging (applies to collaboration application) and the management of the underlying P2P network.
On top lays the *Application Layer* where specific functionalities for applications,tools and services are implemented.

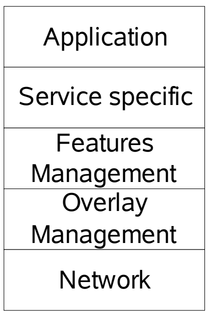


Figure 1: An Abstract P2P Overlay Network Structure (layers)

4.1.2 General Classification of P2P systems: Structured and Unstructured

The term *structured* applies in P2P to the fact that there is some specific control and some deterministic behaviour. Peer identifiers aren't taken randomly, the data which a peer shares is placed in some specified location. Such systems like Chord [4] Tapestry [5], CAN [6], DKS [7] make use of a Distributed Hash Table (DHT). In such a DHT the data object (value) is placed deterministically at peers with identifiers corresponding to that objects unique key. With the DHT's Application Programming Interface (API) objects can be put, retrieved and looked up.
Structured overlay networks introduces such key-based routing which is highly scalable. It is efficient in locating rare items, but produces much overhead in locating replicated items (what is a reason for the high presence of unstructured networks in the Internet).

In contrast, *unstructured* networks have loose rules. Their nature is ad-hoc. A node joining the network doesn't has to know the topology and don't have any prior knowledge. The first unstructured network was Gnutella [8]. It uses a flooding based mechanism to send queries across the network. Each flooding is limited to a certain scope (using a TTL field). A node receiving a query and having a match replies back with a list of all matches to the originating peer. Another unstructured system is Kazaa which is based on the proprietary FastTrack technology [9].
Unstructured ad-hoc networks have often the property called small world [10]. This phenomena is based on the hypothesis that each human in the world can be reached by a short chain of social acquaintance (by a average of 6). Adopted to unstructured networks, there exists relational information between nodes which can be exploited to reduce hop count. This information are vicinity based information. Networks are built ad-hoc and for specific purposes.

Unstructured networks suffer from the problem that they do not scale as well as structured networks: Nodes become readily overloaded when it comes to a higher rate of aggregated queries as in the case of Gnutella.
On the other hand, unstructured overlay networks are efficient on locating replicated data or popular data.

An *unstructured topology is* a overlay network realized with a random connectivity graph where a *structured topology* is a overlay network with a predetermined structure. The latter form a predictable and controllable structure, although it can be fully decentralized.

4.1.3 P2P algorithms: Centralized-, Flooding- and Document Routing Model

This section briefly discuss the types of P2P algorithms. They build the core of a P2P system and describes their type of overlay network.

In a ***Centralized model***, an example is Napster [3] , a central server is holding a directory of shared files and the data which is distributed on the belonging peers. There are 2 services. The first one is the directory service and the second a storage service. Storage service is distributed (the peers) and the directory service is a central server. In principle the peer sends a search query to the directory service and gets a reply with a list of results. The peer therefore can communicate with the storage from peer to peer.

An important drawback is the scalability of the directory service. The centralized service has two crucial properties: it is a bottleneck and a single point of failure. The decisive stage to overcome these issues are scalability techniques.
Anyhow, this model is called a hybrid model because it uses both approaches: a centralized service to look up and a distributed service as its data storage.

The ***Flooding Model*** as used in Gnutella [8] forms a flat or some kind of low-hierarchical (e.g. super peers) random graph. It is very effective in locating highly replicated data. Flooding doesn't guarantee any hit, and for rare items this model is poorly suited. The algorithm is robust against failures and joins/leaves. The system is poorly scalable, since the load increases linearly to the number of nodes in the system [2]. The nodes become overloaded and therefore it doesn't scale well.
Anyway, the flooding model is robust and is a less complex overly than i.e. a DHT based system. Furthermore it's a pretty ad-hoc system where the system it self doesn't matter much about structure.

In the ***Document Routing Model*** each peer is given an ID. A peer can share a document by hashing the documents content and its name (publishing). This hashes forms a Document ID (DID). A peer in the system will route this DID towards the peer with the most alike peer ID. This process is repeated until the closest peer ID is the current peer ID. If a peer now requests a document with a document ID 'DID', the system routes the query towards the peer with the closest ID. This process will be repeated until a copy of the requested document was found. Thereafter, the document is routed back to the originator. On each hop along the route a copy will be kept.

The Document Routing Model scales very good in large scale such as global communities. A drawback is that the document ID must be known before requests are sent. It is a more difficult task to implement search algorithms than in the flooding model. An other common problem is the islanding: if the network splits apart e.g. because of a broken link the communities splits into sub communities which do not know each other.

The following algorithms have implemented the document routing model: Chord [4], CAN [6], Tapestry [5] and Pastry[11] . They share all the same goal which is to reduce the number of hops for locating a document. For more details in peer to peer computing follow up [12].

4.1.4 Distributed Hash Table DHT

A DHT is an infrastructure which enables distribution of an ordinary hash table onto a set of cooperating nodes. DHTs have the important property which consistently assigns random node IDs in a uniformly distributed manner taking the set of numbers from identifier space. Data objects are assigned IDs taken from the same identifier space. A hash function maps object keys, such as a file name, onto the overlay network to a corresponding existent peer in the network: ID = Hash(Key).
The overlay network supports the following API given in Figure 2 API for a structured DHT-based Overlay System

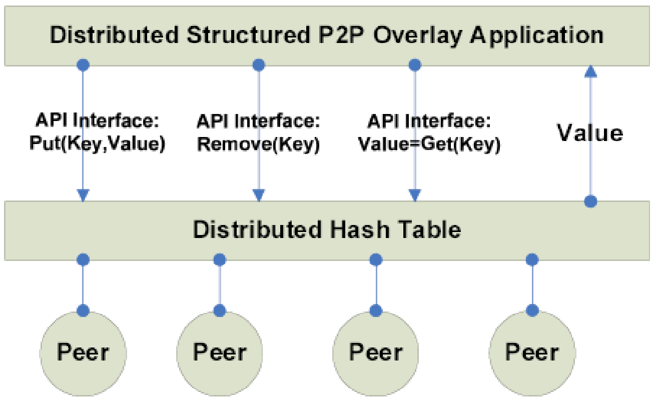

Distributed Structured P2P Overlay Application
API Interface: Put(Key,Value)
API Interface: Remove(Key)
API Interface: Value=Get(Key)
Value
Distributed Hash Table
Peer
Peer
Peer
Peer

Figure 2 API for a structured DHT-based Overlay System

To put a given object onto the DHT, we can make use of the interface put(Key,Value). Key is the key of the object and "Value" is the data object. The operation "lookup" can be achieved by Value=get(Key), which retrieves the data object corresponding to the key. The lookup will initiate routing to the peer holding that data object and get its value.
On the DHT, a key is matched to a ID on the identifier space

In a DHT each peer will maintain a small routing table of its neighbouring peers. Look up routing will be progressively done by locating the data object by locating its closest peer in sense of the peer ID.
In theory, DHTs can achieve routing performance in *O*(log(N)) average on look up, where N is the number of peers in the system.

The underlying network path (the physical one, or e.g. the IP network) between two peers can be significantly different from the path on the DHT-based overlay network. Therefore, the lookup latency in DHT-based P2P overlay networks can be quite high and could adversely affect the performance of the applications running on it.

Many DHT-based P2P lookup approaches heave been proposed. For example: CHORD [4] uses a ring structure for the ID space and each node maintains a finger table to support key query as binary search. Pastry [11] uses a tree-based data structure and the routing table kept in each node is based on shared prefix. P-Grid [13] is based on a virtual distributed search tree. CAN [6] implements DHT using a *d*-dimensional space. These systems are all scalable access structures for P2P and share the DHT abstraction.

Today exists many flavours of DHTs. The original ideas of DHTs are based on 2 ideas: Consistent hashing and the PRR^2 scheme from Plaxton et. Al. PRR^2 is a scheme for efficient routing to the node holding a object while having a small routing table [14].

4.1.5 Distribute K-*ary* Search (DKS)

The DKS is a structured P2P overlay network, implementing the DHT's functionality. DKS is based on CHORD. It uses a virtual k-ary spanning tree, where CHORD uses a binary (2-ary) spanning tree. The height of the DKS spanning tree is $\log_k N$ where N is the number of nodes in the network and k the configuration parameter forming the base of the tree. The lookup is done by following a path in the spanning tree.

The DKS organizes the peers in a circular identifier space and has the routing tables of logarithmic size $(k-1)\log_k N$ (CHORD has k=2).
In DKS the circular identifier space is larger than the number of live nodes. Every node is responsible for some interval on the identifier space. If an object is stored, it will be forwarded to the node responsible for identifier given by the hashed key of that data. The keys are taken from the same identifier space as the node IDs, so there's an implementation of document routing, as explained previously.

The DKS architecture has some important services such as efficient broadcast and multicast of messages (group communication). For more details please refer to [7] .

4.1.6 Common Based Peer Production

The term Peer-to-Peer is not solely a technology-paradigm, as we can find it in social economics, there's an appearance of P2P as well: *Common Based Peer Production* (CBPP) [15] is a new economical model of production coined by Yochai Benker, a professor for law at the University of Yale. The model describes that the creative energy of large numbers of people is coordinated into large projects, mostly without traditional hierarchical organization or financial compensation i.e. Linux or Wikipedia.

The Internet is often used for such ad-hoc collaborating. What brings the development of P2P in strictly technological interest and in the economical interests together is the new paradigm of dynamic collaboration: collaborators are physically dispersed and mobile, they join whenever they want and where they want.

4.2 Web Service and Service Oriented Architecture SOA

Web Service is a very popular paradigm in all economical sectors. One of the most important aspects is the gap-bridging between business concepts and IT concepts. Many companies such as Microsoft, Sun and IBM have quickly discovered the high potential with Web Services. Today, nearly every software vendor has agreed to use the same core standards for WS. The WS is a de facto standard and ratified by the W3C [16].

Service Oriented Architecture (SOA) changes the process of designing, developing and deploying software. The SOA defines an architecture for *loosely coupled software services*. In the SOA there are three different individual roles:

1. Service Provider: Implements the service and provide it in the internet
2. Service Consumer: Searches and uses provided services
3. Service Registry: Enables search of services and holds information of service provider

Existing software can be converted into services, even monolithic and inflexible applications can be replaced by SOA architecture applications. There is no coupling between owner and consumer of the service.

4.2.1 Definition Web Services

Web Services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located and invoked across the Web. Web Services perform functions, which can be anything from simple requests to complicated business process.
Once a Web Service is deployed, other applications (and other Web Services) can discover and invoke the deployed service"
IBM Web Service tutorial [17]

An other definition:

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.
W3C Working Group[16]

Web Services provides an interface for distributed applications to communicate with each other. It defines a set of protocols that enables applications to publish, search, provide and consume a service. All protocol are based on XML.

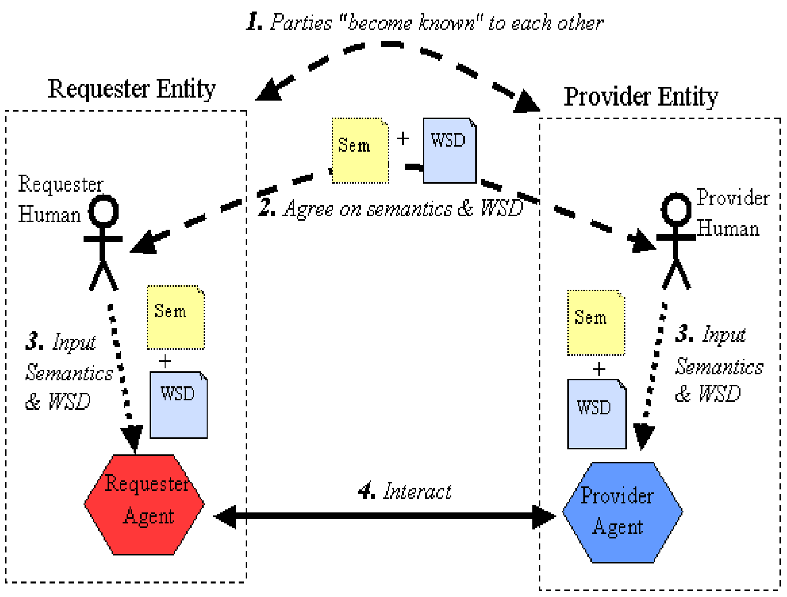


Figure 3: General Process of Engaging a Web Service [18]

The provider exposes a service to the environment. It might be a person, an organization or a company which publishes services at a known place. A consumer can therefore find the published service; the entities consumer and provider become known to each other. Entities agree on a *semantic* which enables their message exchange (mechanics), interpreting and acting on these messages.

The mechanics of a Web Service message exchange are described in the so called *WSD* or Web Service Description (using XML). It is a machine processable specification of Web Service interface. It defines the message formats, data types, transport protocols, and transport serialization formats being used between consumer and provider (depicted in Figure 3). Upon these information a consumer can consume the providers service.

In general a Web Service interaction can be seen as the following:

1. Client queries registry to locate services
2. Registry refers client to WSDL document
3. Client accesses WSDocument
4. Client processes WSDL, provides information to use Web Service
5. Client sends SOAP message request
6. Web Service returns SOAP-message request-response

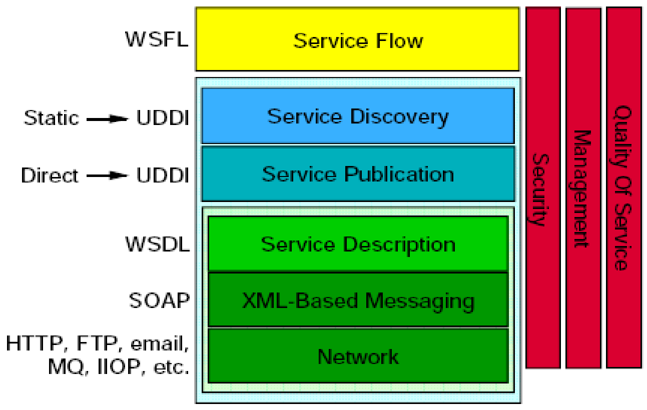


Figure 4: WS interoperability stack

4.2.2 Messaging SOAP

"The Simple Object Access Protocol is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment " [18].
The SOAP messaging is XML messaging. It provides a flexible means to communicate between applications. Because XML is not bound to some programming language or operating system, messaging can be performed independently of these. SOAP is a standard way to structure the messaging in Web Services.
SOAP is under standardization of W3C's XML protocol working group, after Microsoft, IBM, Ariba and some smaller companies had submitted SOAP in the year 2000.

The SOAP message consists of an envelope containing a optional header and exactly one body. The header contains blocks which indicates how the message must be processed. These can be authentication credentials, routing information or it can be a transaction context.
The body in the SOAP envelope contains the application payload. The bodies content is pure application specific and not part of the SOAP-specification.

SOAP fits in WS as a standardized packaging protocol on top of the WS technology stack, above the network and transport layers. As a packaging protocol, SOAP doesn't care about what transport protocol is used. This makes SOAP flexible in where and how it is used.
SOAP-over-HTTP is the far most used transport and the SOAP specification even gives special treatment for SOAP on HTTP. Despite that HTTP is pervasive on the Internet, SOAP can be transported as well on SMTP, POP3, FTP and on many other transport protocols.

SOAP implementations are:
-Apache SOAP (http://xml.apache.org/soap/)
Open source Java implementation of the SOAP protocol; based on the IBM SOAP4J implementation.
-Microsoft SOAP ToolKit (http://msdn.microsoft.com/soap/default.asp) COM implementation of the SOAP protocol for C#, C++, Visual Basic, or other COM-compliant languages.
-SOAP::Lite for Perl (http://www.soaplite.com/) Perl implementation of the SOAP protocol, written by Paul Kulchenko, that includes support for WSDL and UDDI.

-GLUE from the Mind Electric (http://www.themindelectric.com) Java implementation of the SOAP protocol that includes support for WSDL and UDDI

4.2.3 Service Description WSDL

Web Service Description Language describes the data type information for message request and message response, how they are bounded to a transport protocol and how services can be invoked. All these information are specified in the WSDL specification [19].

In a nutshell, WSDL is a contract between the service provider and requester, using a XML grammar. As in SOAP, WSDL is language- and platform-independent. WSDL is primary used to describe SOAP services (but it isn't limited to that).

WSDL 1.1 (submitted by Microsoft, IBM, Ariba and many small companies) and 2.0 is a W3C candidate recommendation, meaning that it's a document that W3C believes has been widely reviewed and satisfies the W3C Working Group's technical requirements.

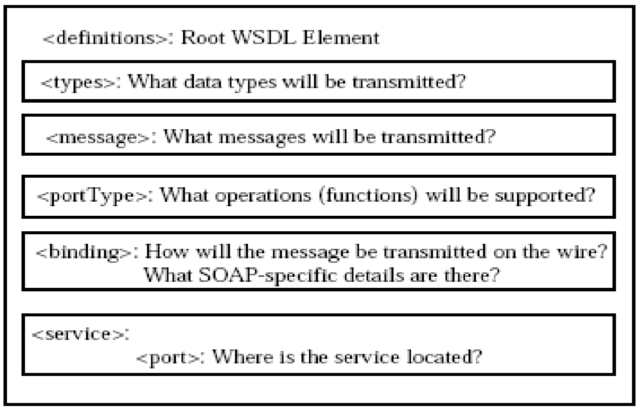

<definitions>: Root WSDL Element
<types>: What data types will be transmitted?
<message>: What messages will be transmitted?
<portType>: What operations (functions) will be supported?
<binding>: How will the message be transmitted on the wire? What SOAP-specific details are there?
<service>:
<port>: Where is the service located?

Figure 5: The WSDL Specification

The WSDL specification can be split into 2 definition parts: The *service interface definition* and the *service implementation definition.*

The *Service Interface Definition* is contains reusable parts and is expressed by Binding, PortType, Message and Type.

Types: describes all the data types used between consumer and service provider.

Message: It describes a one-way message which can be either a response or a request. It defines the message name and it can contain zero or more parts. Parts are usually some parameters or return values.

PortType: It combines multiple messages to form a one-way or two way request response operation. Most commonly used in SOAP is the combining of a request message and a response message in a single request/response operation. Operations describes actions supported by the messages.

Binding: This element describes how the service is specifically implemented on the wire.

The S*ervice Implementation Definition* describes how a service is implemented by a service provider *and it* is expressed by the *Service* and *Port.*

Service: Specifies the location of the services through *Ports* (or end points). It contains a documentation element to provide human readable documentation

The two definitions can be combined in one document or be a part in two documents. This separation enables interface re-usability. Implementation and interface can be treated dependent from each other.

4.2.4 Discovering and Publishing Services

An open environment allows to choose what service and when it should be consumed. To be able to search for a service, services must been announced or published. This might be a a directory service. I could look up in that directory for a service best suited to my needs, fetch the description and consume that service.

UDDI is a technical specification for describing, discovering, and integrating Web Services. A definition from the OASIS UDDI Specifications TC - Committee Specifications [20]:

> „*UDDI Version 3.0, an OASIS Standard, builds on the vision of UDDI: a "meta service" for locating Web Services by enabling robust queries against rich meta data. Expanding on the foundation of versions 1 and 2, version 3 offers the industry a specification for building flexible, inter operable XML Web Services registries useful in private as well as public deployments* "

UDDI 1.0 was originally announced by Microsoft, IBM, and Ariba in the year 2000. Since then the UDDI.org initiative has grown up to more than 300 companies. In 2001 Microsoft and IBM launched the first operational UDDI site which was shut down end of 2005. Later on, in 2001, UDDI.org announced version 2 with extended features. After completion of version 3.0 UDDI.org submitted it to OASIS to evolve into formal standard. Nowadays UDDI 3.0 is an OASIS standard. UDDI is not part of the standardization effort by W3C.

UDDI is one of the core WS standards. It is designed to be queried by SOAP messages to retrieve Web Service description documents (WSD), which contain all information to consume a service. The UDDI defines data structures and a API for publishing and querying the registry.

The information in a UDDI can be in:

- White pages: contain general contact information about the entity
- Yellow pages: contain classification information about the types and location of the services the entry offers
- Green pages: contain information about the details of how to invoke the offered services (technical data regarding the service)

UDDI is based on a common set of standards, including HTTP, XML, XML-Schema and SOAP.

4.3 Grid Service

Grid Computing is an emergent technology in the world of distributed computing. Grid Computing is the next logical step in networking. Like in the World Wide Web, Grid Computing allows people and machines to share files over the Internet. Grid computing enables sharing machine resources like computational power and storage capacity over the Internet. A definition by IBM [21]:

> *„Grid computing allows you to unite pools of servers, storage systems, and networks into a single large system so you can deliver the power of multiple-systems resources to a single user point for a specific purpose. To a user, data file, or an application, the system appears to be a single enormous virtual computing system."*

Ian Foster aka father of the Grid, senior scientist in the Mathematics and Computer Science Division at the Argonne National Laboratory Chicago, defines a 3 point check list what specifies a Grid[22]: A Grid is a System that

1. coordinates resources that are not subject to centralized control …
2. … using standard, open, general-purpose protocols and interfaces
3. … to deliver non-trivial qualities of service.

Resources and users are in different control domains and the Grid integrates them. The open standards and general-purpose protocols builds a collection of heterogeneous systems. In the anatomy of the Grid [23] Foster points out that the real problem underlying the Grid concept is the coordinated resource sharing and dynamic, cooperative, multi-institutional collaborating. These emerges sharing rules, called *Virtual Organizations VO*.

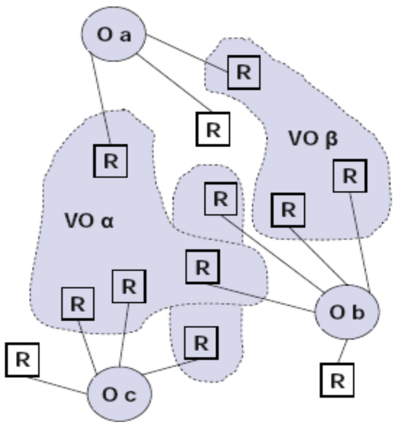


Figure 6: VO sharing resources R from organizations O

Such VO enables high performance and throughput by aggregating resources from different organizations together. VOs are dynamic heterogeneous federations which share processing power, data and the security infrastructure. In the view of what forms a VO we can think of organization which enforce security rules and implement policies for resources utilization and priorities of using them. VOs can be companies, organizations, institutes, a collaborating compound of the previous named and so on. These might also be projects which are existent over long time or only for short time.

Grid virtualizes heterogeneous geographically disperse resources. Files and Data Bases can seamlessly span over the globe, capacity can be improved for data transfer rates.

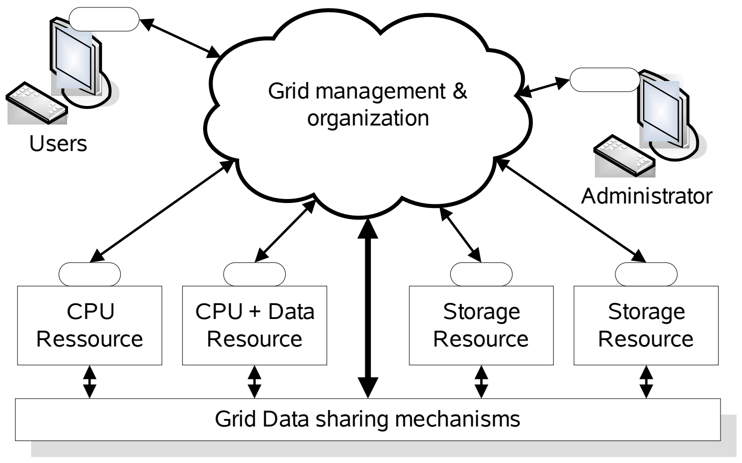


Figure 7: A simple Grid on a local organization.

The IBMs redbook "Fundamentals in Grid computing" [24] boils the principles of Grid computing in a business scope down:" if you want to meet customer requirements to better match within the Grid computing, you should keep in mind the reasons of using Grid computing: exploiting underutilized resources and parallel processing power".

The aspect of reliability and management in IT infrastructure exploits new business possibilities within the Grid environment. Reliability in IT infrastructure is achieved nowadays by hardware redundancies such as multiple CPU, storage striping (RAID) or gasoline generators for electricity blackouts. With the Grid paradigm a relatively inexpensive and geographically dispersed redundancy can be achieved: The blackout in Moscow doesn't affect the city of Stockholm.
Using „automatic computing" allows an automatically healing in the Grid. The vision is clear: where reliability is done today in hardware, it will be achieved in future in Software.

In the management perspective of IT infrastructure, the virtualization of the resources in the Grid will allow us to better manage large and disperse, heterogeneous system [24].

Where Grids are used:

- In the financial services industry, Grid computing can be used to speed trade transactions, crunch huge volumes of data, and provide a more stable IT environment in a mission-critical environment that doesn't tolerate much downtime.
- Government agencies can use Grids to pool, secure, and integrate vast stockpiles of data. Many civilian and military agencies need the capabilities of cross-agency collaboration, data integrity and security, and fast information access across thousands of data repositories.
- Companies involved in the life sciences, such as those that do genome research and pharmaceutical development, can use parallel and Grid computing to process, cleanse, cross-tabulate, and compare massive amounts of data. Faster processing

means getting to market faster, and in those industries, a slight edge can be the deciding factor.

The *Grid Architecture* as proposed by Kesselman, Tuecke and Foster's anatomy of the Grid [23] catalogues the components of a Grid system which is an extensible, open architectural structure.

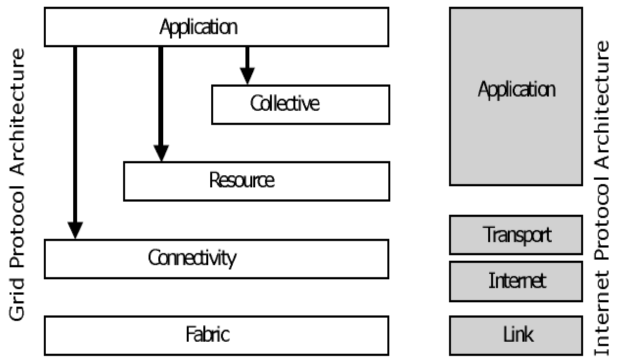


Figure 8: Grid Architecture and its mapping to Internet Protocol

Fabric Layer: The layer is the local control of the actual resources (e.g. hardware), underlying the Grid system. This can be computers, supercomputers, storage, clusters or sensors.

Connectivity Layer: Defines the communication and security. The layer enables fabric layer resources the exchange of data between them. Authentication protocols provide cryptographically mechanisms for verifying users and resources. The fundamental Internet protocols such as HTTP, TCP/IP, DNS fall into this layer.
It describes what authentication solutions characteristics for VO should be possible like *single sign-on*, *delegation of credentials*, i*ntegration with various local security solutions* and *user-based trust relationship*.

Resource Layer: It enables to manage the local resource individually. It is builds on top of the Connectivity layer and defines protocols, APIs and SDK for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. The resource layer calls operations on the Fabric layer to interact directly with the local resource. The layer doesn't care about global state.
Management protocols are used to manage the access and control of the shared resource. These protocols also are responsible to accomplish it's organizational policy of what operation can be taken out by who. GT4 adopts a set of protocols like GRIP, GRAM, GridFTP and LDAP [23].

Collective Layer: Coordinates multiple resources. It manages a collection of resources and make them working together to solve a common task. The layer provides services such as:

- *Directory Service*: discovering VO's resources and their properties
- *Co-allocating, scheduling and brokering:* lets VO participants allocating and scheduling resources for specific purposes. A program we would like to run (called

a job) will be allocated by discovering resources through a directory service and will allocate the needed resource for that job

- *Monitoring an diagnostics services:* The VOs resources can be monitored and probed e.g. if any attacks, failure or overload happened.
- *Data management service:* Jobs will require data to work on. Therefore the data management keeps track of these data and transfer them to the resource which needs them within the VO.
- More services are: *Workload management systems and collaboration frameworks, Data replication services, Community accounting and payment services, Collaborator services*

Application Layer: The top layer is the virtual organization environment to execute applications. The Layer does not has to interact with the Collective Layer but can also directly interact with the resource and connectivity layer.

4.3.1 Open Grid Services Architecture

OGSA is a specification which defines the overall structure and services which can be provided in the Grid. The specification defines a common, open standard architecture for Grid-based applications. The 'open' in the OGSA defines interoperability and the artefact of standardisation should guarantee the portability of OGSA implementations.
OGSA is developed by members of the OGF Open Grid Forum www.ogf.org formerly called GGF Global Grid Forum.

The OGSA adopts Service Oriented Architecture SOA. Everything is a Web Service which is the main groove in the architecture – various resources become available as a Web Services. The OGSA requires stateful resources, more precisely it uses the Web Service Resource Framework WSRF, which will be explained later.

The OGSA defines a frame work which strongly uses the component paradigm. The components can be expressed as capabilities which offers functionalities respectively services to the desired needs. These capabilities or services are not standardized, they are rather informative for adoption in a particular implementation. The architecture is not a layered or kind of object-oriented architecture. The following services are identified by OGSA and should be encountered in a Grid system:

Execution Management Service are concerned with the problems of instantiating, managing, and completion of work units. OGSA data services are concerned with the movement, access and update of data resources. It is as well concerned with data replication and data consistency.

Resource Discovery and Management Services: In an OGSA Grid there are three types of management which involve resources: Management of resources themselves (e.g., rebooting a host), management of the resources on Grid (e.g., resource reservation, monitoring and control), management of the OGSA infrastructure, which is itself composed of resources (e.g., monitoring a registry service).

Security services are to facilitate the enforcement of security-related policies within a virtual organization. Security at a high level is authentication, delegation, single sign-on, privacy, confidentiality, integrity and so on. Security is one of the most challenging parts in Grid and can be specifically been described in a Grid security model.

Self-management is an automated process which reduces the cost and complexity of owning and maintaining an IT infrastructure. In such an automated managed environment, the whole infrastructure, including hard- and software, becomes optimized, self-healing and self-configuring.

Information Service provides efficient production of, and access to, information about the Grid and resources. This includes as well status and availability of a particular resource.

Context Management Services manages the usage and access of resources for users. It optimizes resource utilization based on resource requirements.

4.3.2 Web Services Resource Framework

The WSRF is a joint effort by the Grid and Web Services communities. The WSRF [25] is an extension to the Web Services and specifies stateful Web Services.
Operations in a Web Service might have values as parameter or results. To be able to remember such value after an operation has finished, it must be stored in memory. The memory might be simple variables, data structures or data bases and so on. A Web Service can have access to many different resources. This what is referred as state, a well defined way to store and access values on the service provider side. Stateful resources inherently enables high complexity and transactions for Web Services.
Note that stateful resources appear in several computing contexts. Stateful resources are a major focus of Grid Computing, as in the Open Grid Service Infrastructure 1.0 OGSI [26].

The State is not kept in the Web Service; State is kept in a *resource* while the Web Service itself is state less. They are well separated and together they form the Web Service Resource (**WS-Resource**).

Addressing WS-Resources is specified in the **WS-Addressing** specification. It defines a construct called *endpoint reference* which allows to address Web Service endpoints. It is a XML construct which includes an URI pointing to the corresponding Web Service. The resource itself can be identified by an resource identifier. The latter is called WS-Resource-qualified endpoint reference.

Resource Properties are the actual data items within the resource. An example of a resource property might be „File name“, „Size“ , „Descriptor“ and the like. The resource properties are generally used to store *service data values* (reflects service properties like operation results, runtime information), *meta data about values* (like who accessed last, what was it's changing time) and *state management information* which manages the resource as a whole (e.g. manages its lifetime).

The WSRF specification is a collection of four different specifications which relate to the management of the Web Service Resource:

- WS-Resource Properties
- WS-Resource Life Time
- WS-Service Group
- WS-Base Faults

please refer to the WSRF [26] for detailed information.

A related specifications within WSRF is **WS-Notification** which describes how a Web Service can be configured as a *notification service* where clients can subscribe to it and become a *notification consumer*.

4.4 Grid Service versus Web Service

Although Grid Services are implemented using Web Services technology, there is a fundamental difference between a Grid Service and Web Service.
A Web Service addresses the issue of discovery and invocation of persistent services. A Web Services Description Language (WSDL) compliant document points to a location that hosts the Web Service.
A Grid Service addresses the issue of a virtual resources and the management of state. A Grid is a dynamic environment. Hence, a Grid Service can be *transient* rather than persistent. A Grid Service can be dynamically created and destroyed, unlike a Web Service, which is often presumed available if the corresponding WSDL file is accessible to clients.

Web Services also typically out live all their clients. This has significant implications for how Grid Services are managed, named, discovered, and used. The OGSA model adopts a *factory design pattern* to create transient Grid Services. Thus, an OGSA Grid Service is a *potentially transient Web Service* based on Grid protocols using WSDL

4.5 Grid Software

The OGSA is a reference architecture for open and interoperable implementations of Grid systems. In this section we are going to see what a real Grid software is composed of and how OGSA is adapted. In the remainder of this section I will discuss existent products and their properties.

Distributed Grid Management
This component keeps track of available resources and assigns Grid jobs. It measures the utilization rate and capacities of nodes in the Grid. The management of the Grid must be a scalable and highly available component. To achieve that it must be achieved in a distributed manner. The primary job is to collect statistical information in the Grid in a distributed way, using an aggregation approach.
The IBM redbook "Fundamentals of Grid Computing" [24] conceptually decompose a Grid Software in the following components:

Donor software
A machine, e.g. a PC, would like to share its computational power. It therefore installs a software making it a potential member of the Grid system. Such that a machine can join a

Grid, different security aspects has to be performed: Establish and proof relationship and identity, obtain a member certificate in the Grid, login in the Grid and so on.

Job submission software
The software used to submit jobs into the cloud. Any member machine, node, in a Grid can use such software to submit jobs into the Grid. However, special dedicated machines like submission-nodes or submission-clients are chosen to perform the task.

Schedulers
Mostly all Grid system do include schedulers. They organize the job queuing in the Grid. A simple example is the round-robin approach where each after another gets a job to process. Other approaches are i.e. priority queuing or policy based queuing. Schedulers usually act on the immediate Grid load.
Schedulers might be organized hierarchical: meta scheduler/low level scheduler scheme: meta scheduler submits a job to the cluster scheduler, cluster scheduler allocates next suitable node for the job.
More advanced schedulers monitor the progress of scheduled jobs. They manage the overall work flow (outage of jobs, infinite looping, jobs have different completion code). The reservation of resources can be achieved in a calender based system.

Communication
Grid Software might include software to help jobs communicate with each other. An input of one job might be the output of another job. They might not reside on the same resource and hence need to communicate with each another.
One of the open standard which enables this communication is called MPI, Message Passing Interface. MPI and variations of it are often included as a part of the Grid system. The most common protocol used is SOAP.

Observation and measurement
The donor software usually includes some facilities to monitor the hosts load. These are also called load sensors. Facilities might be explicitly built in or can be used as offered by the hosting operating system. The measurement of CPU (process usage) and storage usage is measured but also job progress is monitored. These enables a predictable concept of what a job resource needs are, allowing better scheduling.
There exist different Grid architectures to fit the specific business needs. Some Grids architecture are thought of take computational resource power into its main objective where others are targeting collaboration problems between different organizations. This means that the selection of the Grid type has a direct impact on the design of the solution.

There are several Grid Software available: Globus Toolkit 4, GT4 from the Globus Alliance is a freely available. Sun's N1 Grid Engine and IBM's Grid Tool Box are commercial products. gLite, a middle-ware for Grid computing from the EGEE at CERN (Enabling Grids for E-ScienceE). GRIA is Grid Resources for Industrial Applications, aimed for business users.

4.5.1 Globus Toolkit 4 and GRAM

Globus Toolkit, GT, is an OGSI implementation of the Globus Alliance[27]. Some of the core components of GT4 are:

- GRAM: Globus Resource Allocation Management. It is the heart of GT Execution Management. It provides services to deploy and monitor jobs on a Grid.
- GSI: Grid Security Infrastructure, provides Authentication and Authorization, Credential Delegation, Community Authorization for VO and credential Management
- MDS: Monitoring and Discovery System, provides Index Service to aggregate resources of interests in a VO, Trigger Service (same as index service but actions might be triggered based on some data).
- Data Management: GridFTP (optimized for data transfer between hosts), Reliable File Transfer Web Service (RFT).

APIs and command line utilities are provided with the software. For more information see [27].

Grid Resource Allocation Management GRAM 4
GRAM is a set of Web Service components providing a single standard interface for using remote resources. The interface allows the bidirectional communication between resource and clients which utilize that resource.
The hourglass model illustrates the GRAM as the neck in the model.

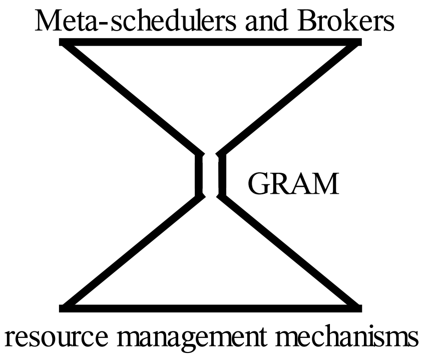


Figure 9: The hourglass model of GRAM

Meta schedulers and brokers might allocate on a higher level the GRAM. These are applications and higher order services which sit above GRAM. Below GRAM are local control and access mechanisms.

In the scope of Grid jobs, GRAM allows clients to submit, monitor and control jobs remotely. Four basic services that are provided by GRAM are

- MJFS: Managed Job Factory Service,
- MJS: Managed Job Services,
- FSFS: File Stream Factory Services,
- FSS: File Stream Service,

Jobs are executed on the host machines as local users. GSI authenticates users and resources. There are mechanism for mapping Grid users with local users, and for credential delegation.

A job is specified through the *Resource Specification Language* RSL. The XML based language models the GRAM capabilities and describes a job for execution. RSL extensible for more complex expressions.

We consider GRAM as the component designated in our design as the resource allocation manager for single resources. Therefore each resource will have exactly one GRAM component as the interface between resource and client.

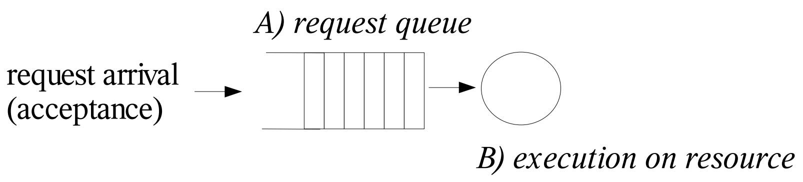


Figure 10: FIFO discipline on a resource with GRAM

Job submission on GRAM are in a FIFO discipline. On a resource, a submitted job means that GRAM has accepted the job and executes it within the queue (as in a batch system).

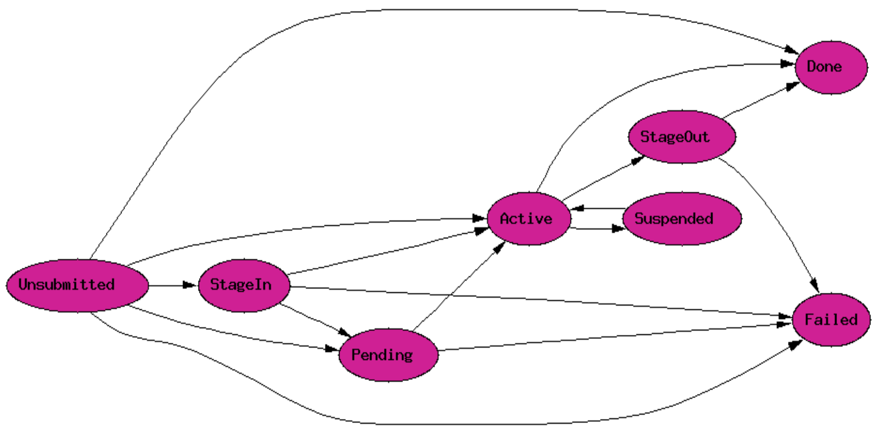


Figure 11: Different states of a job in the GRAM scheduling model

A job has different states as depicted in Figure 11. A job might need to stage in files beforehand and also might have to stage out results. To interact with a GRAM, a client uses the GRAM API. In essence, a job request in GRAM is a request to create a job process, expressed in the supplied Resource Description Language RSL. The Request guides:

- resource selection, *when and where should process be created*
- job process creation, *what job process should be created*
- job control, *how should job processes being executed*

In our work, we focus on the resource selection by routing requests. Whenever a client submits a job to the Grid, the selection of the resource will be decided by a component called the routing component. We use the GRAM as an abstraction, or more as a example, to specify our model.

4.6 Engaging Grid and P2P

P2P and Grid are both focusing on the coordination and sharing (pooling) of resources within distributed communities. Some P2P systems have been referred as a Grid. Even Grid and P2P solving similar issues, there are significant differences in communities, incentives, applications, technologies, resources and achieved scale [28].
Grid systems are used for intensive computations and data manipulations. To realize authentication requirements and sharing policies among the parties in the VO, a centralized approach is employed. The centralization makes Grid system inherently unscalable.

On the other hand, P2P systems are mainly driven by file sharing communities, despite that much research in P2P systems is going on. Anonymity is highly valued and trust assumption simply doesn't exist.
The evolution of file sharing P2P systems [29]:

1. Generation: Server Client (Usenet, Napster)
2. Generation: Decentralized (Gnutella, FastTrack, Edonkey, BitTorrent...)
3. Generation: High Anonymity with non-direct and encrypted (Waste, Ants, Mute, I2P)
4. Generation: stream over P2P

P2P systems do not have any centralized requirements. This makes P2P systems highly scalable, fault tolerant and self managing.

Engaging the elements of P2P and Grid computing we can solve the problems, which occurs in Grid systems, that address scalability and failures by using self-configuring protocols such of P2P systems.

5 Survey of Load Balancing in Distributed Systems

A small survey for different approaches gives us an overview of existing solutions and source us with ideas and inspirations. There are 4 different types of load balancing considered in the survey:

- dynamic load balancing for distributed and parallel object-oriented applications in a P2P system;
- Load movement in a structured P2P network;
- Grid Load Balancing using intelligent agents and multi agent system;
- Messor: Load distribution based on the ant colony metaphor

5.1 Active Object Migrations

In [30] Javier Bustos and Denis Caromel present an algorithm to balance load of Java Virtual Machines (JVMs) on the Grid middle-ware ProActive [31]. ProActive is an open source Java middle-ware, which aims to achieve seamless programming for concurrent, parallel, distributed, and mobile computing, implementing the active-object programming model.
An Active Object has an active thread and is composed of a body and a standard Java Object (also called passive Object). The body is responsible for receiving method calls and storing them in a queue. The thread chooses then a method in the queue and executes it as the standard Object.

In ProActive, active objects are accessible remotely via method invocation. Method calls with active objects are asynchronous with automatic synchronisation. This is provided by automatic future objects. As a result of remote methods calls, synchronisation is handled by a mechanism known as wait-by-necessity. Wait-by-necessity (WbN) is an active object request which has not yet been served, and it waits for the responses thus reflects a longer execution time. By reducing the WbN time, performance can be improved [32].
ProActive provides a way to move any active object from any Java Virtual Machine to another, through migration. Active Objects can be migrated through a local or external (by an agent) call. Any active object can be migrated. If there are some passive objects referenced with it, they are migrated along. All Objects must be serializable to be migrateable [31].

The balancing objective is to reduce the WbN time to improve the overall execution time. Through the migration of active objects on computational nodes with better resources, execution time will be reduced due to a smaller WbN time and finally to a better overall performance.

A P2P infrastructure is employed where peers have to maintain a list of neighbours (known nodes). A peer joining the network has a list of potentially network peers which it. A requested peer will accept the joining node with a certain probability and if accepted becomes it's acquittance. The node which accepted it forwards the request message to its acquaintances so that the new node gets more possible acquaintances [32]. This resembles the Gnuttella protocol. Nodes can communicate with their acquaintances only. A node, also called a computational node, is a JVM in the P2P overlay network.

The aspects of the load balancing algorithm relies on two approaches:

- if a node is overloaded, migrate objects to a less loaded node;
- if a node is under-loaded, steal work from other nodes which are more loaded than

In the first approach, an overloaded node will send a request to a random subset of its acquaintances. Only under-loaded nodes satisfying a rank criteria will respond. The node migrates an active object to the first node in response. Using the first node in response scheme maintains the property of keeping active objects in vicinity and reduces communication latency.

In the second approach, the extension of the first approach, nodes become active in stealing work from it's acquaintances. An under-loaded node sends to a randomly chosen acquaintance a stealing-request. If the requested satisfy a rank criteria, it will return a active object to the requester. This will cluster active objects on high performance nodes, meaning that nodes with a better resources will have more active objects than nodes with less performance.

An experimental evaluation shows that the algorithm scales well. The authors simulated the algorithm on a P2P network with up to 8000 nodes. To verify it does scale well, experimental tuning of algorithm parameters were made. The interesting metrics for scaling capabilities are how many migration are performed and how big the ratio between the optimal distribution of the objects and the experimented distribution of the objects are. However, the authors conclude out of their tests that by having a low number of links per node and well tuned algorithm a near optimal-distribution is reachable even for large scale networks.

5.2 Load movement in a P2P structured network

The difference between neighbouring knowledge and partial knowledge is that the latter one is knowledge of the partial system, where the first one is knowledge in the vicinity of a node. Partial knowledge is more appropriate in structured network since structure is known, whereas in unstructured networks no assumption of the structure of a network can be made.

Partial knowledge can be exploited such as in [33] by using the resource routing model. In this work the authors state in their summary:

> *We propose an algorithm for load balancing in dynamic, heterogeneous peer-to-peer systems. Our algorithm may be applied to balance one of several different types of resources, including storage, bandwidth, and processor cycles. The algorithm is designed to handle heterogeneity in the form of (1) varying object loads and (2) varying node capacity, and it can handle dynamism in the form of (1) continuous insertion and deletion of objects, (2) skewed object arrival patterns, and (3) continuous arrival and departure of nodes [...]*

In a structured P2P system, a unique identifier is associated with each data item and each node in the system (DHT). The identifier space is partitioned among the nodes, and each node is responsible for storing all the items that are mapped to an identifier in its portion of the space.

The nodes represent the processing units, the ones which take out the work. The data items, on the other hand holding meta information, that is information like the memory size or processor time needed to serve a task.
The DHT is the basic core of the load balancing. The authors developed algorithms that completely relies on the implementation of the underlying DHT without making any programmable change to it. The authors are using CHORD[4] in their example. CHORD was one of the first which proposed the notion of virtual servers to improve node imbalance.

Their algorithms use the concept of Virtual Servers. A virtual server represents a peer in the DHT; that is, the storage of data items and routing happen at the virtual server level rather than at the physical node level. A physical node hosts one or more virtual servers. Load balancing is achieved by moving virtual servers from heavily loaded physical nodes to less loaded nodes. In other words: the load is balanced by reassigning the set of region to an other node. A set of region are the data items under the obligation of a virtual server. Because in a DHT the data items must preserve their identity in respect to be routed to, the concept of virtual server was introduced. The set of attached data items to a region follow without ever changing their identifier respectively their place in the identifier space.
The objective of the load balancing is to minimize the imbalance on the DHT by satisfying the requirement of minimizing the amount of the load moved.

The basic idea of the load balancing algorithm is to store load information of the peer nodes in a number of directories which periodically schedule reassignments of virtual servers to achieve better balance. Thus it essentially reduces the distributed load balancing problem to a centralized problem at each directory. The algorithm has two schemes:

- many-to-many: periodical load balancing of all nodes
- one-to-one: emergency load balancing for an overloaded node

In the first scheme, nodes report their load to a random chosen directory out of a subset of two (to the one with fewer 'node reports' to reduce node imbalance among directories). The directory schedules transfers of virtual servers for the nodes. Transfers are scheduled in large batches. Because computing a reassignment for virtual servers to minimize the maximum node utilisation is NP-complete, they used a simply greedy algorithm to find a approximate solution.
If the node becomes overloaded an immediate load movement takes place. The second scheme is applied in the emergency load balancing, where a overloaded node reports to a directory and immediate gets load transferred to reduce its load.

Performance of the algorithm has been evaluated by the following metrics:

- Load movement factor under different system load
- 99.9th percentile node utilization for different load movement factor

The Load movement factor is defined as the total movement cost incurred due to load balancing divided by the total cost of moving all objects in the system once.
The 99.9th percentile node utilization defined as the maximum over all simulated times t of the 99.9th percentile of the utilizations of the nodes at time t. The Utilization of a node is its load divided by it's capacity.
The experimental evaluation consists of 4096 fixed nodes, 12 virtual servers per node and 16 directories and an average number of objects: 1 million. Different patterns have been experimented with to measure those performance: Non-uniform object arrival patterns,

node arrival and departures where node arrivals are modelled as a Poison process and where the lifetime of an object is drawn from an exponential distribution.
The simulation results show that the algorithm is effective in achieving load balancing for system utilizations as high as 90% while transferring only about 8% of the load that arrives in the system, and performs only slightly less effectively than a similar but fully centralized balancer.

5.3 Grid Load Balancing using intelligent agents

The agent paradigm is known for its great ability for modelling complex software systems. The work [34] focuses on Grid load balancing with intelligent agents and multiagent approaches. These approaches were used to schedule local Grid resources and do global Grid load balancing.
A Grid resource can be a multiprocessor or a cluster of workstations. An agent is at the Grid level a presentation for a Grid resource offering services and a high performance computing power. Agents are the high-level abstraction of a Grid resource. Each agent consists of 3 main layers, from bottom to top: communication, coordination and local management layer. The latter performs functions of an agent for local Grid load balancing. The coordination layer treats requests and organizes the local knowledge. The communication layer enables to interact with other agents.

The agents are organized in a hierarchical structure and have a cooperative behavior to advertise services and discovering services by means of P2P mechanisms.

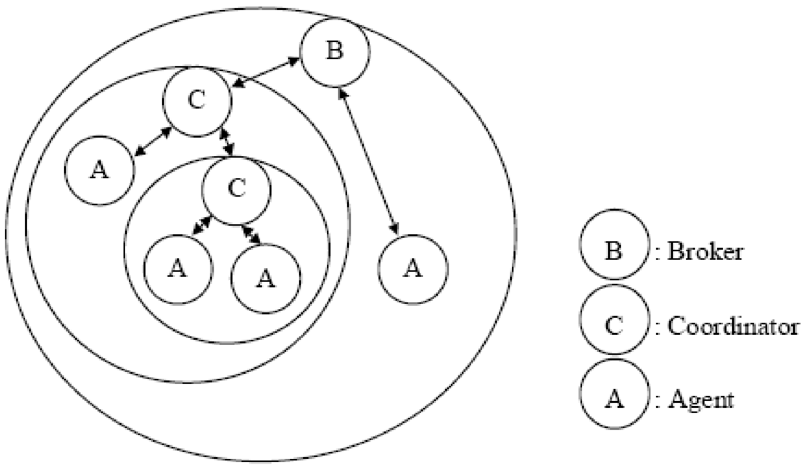


Figure 12: Hierarchical structure

Agents contact each other through their identity, which is like addressing. The broker is On top of the hierarchy. A coordinator is the head of a sub-hierarchy. Brokers and coordinators are also agents with the difference that they are in a special position. Despite their position, they have all the same functionality.
The agent hierarchy can represent an open and dynamic system, meaning agents can join the hierarchy or leave the hierarchy at any time. The hierarchy exists only logically and each agent can contact others as long as it has their identities.

The authors point out that the hierarchical model addresses only partly the scalability. The more agents in the system the higher the system activities. Thanks to the hierarchy, these activities are processed in local domains and thus allows avoiding bottlenecks. Even the hierarchical model is known for good performance in large scale networks [DNS], they have not made any experimental verification of scalability.

The agents use the PACE performance prediction engine. PACE [35] is a tool set for performance prediction in Parallel and Distributed Systems. The evaluation engine combines the PACE resource model and application model at run time to produce evaluation results, e.g. estimation of execution time.
The resource model includes performance related information of the hardware on which the parallel program is executed, specifically on which Grid resource, whereas the application model includes all performance related information of the parallel program, e.g MPI (message passing interface) or PVM (Parallel Virtual Machines) program running on this Grid resource.
The result can be used to feed the algorithms with the necessary information. The algorithms are developed in two different scopes:

- Local Grid Load Balancing;
- Global Grid Load Balancing;

In the first, a local Grid resource is considered to be a cluster of workstations or a multiprocessor. The authors show 2 different algorithms for local Grid load balancing, meaning balancing the load in the scope of a cluster or multiprocessor. The first algorithm is first-come-first-served. The function of the agent local management is to find the earliest possible time for each task to complete, in respect to the sequence of the task arrivals. The main problem with the algorithm is that when the number of Grid resources increases, the scheduling complexity increases exponentially. In this algorithm the ordering of tasks is based on the arrival of tasks.
Considering reordering could improve performance, but also increase complexity. In their second algorithm the authors consider an iterative heuristic algorithm to overcome that issue. The problem of local Grid scheduling became now an optimization problem. Their algorithm is a genetic algorithm where the goal is to minimize the latest completion time when all tasks are considered together. The genome consists of the order in which the task must be executed, and a mapping which allocates the tasks to a host in the cluster. The optimization is measured by a fitness function in each generation, to steer the outcome in the optimum direction. The aim is to find a near optimum solution. For more details about genetic algorithms, read [36].

An advantage of the evolutionary algorithm is that it is adaptive to changes in the system. It absorbs changes such as addition or deletion of tasks or changes in the number of hosts.
The two algorithms are implemented by an agent and they can be easily switch from one to the other. The algorithm can be best used in a local Grid resource since it offers a fine grained adjustment. However, this algorithms can not be employed for a large scale, since complexity increases exponentially with the number of hosts.
The second deals with the Grid load balancing. The problem that is addressed in this algorithm is how the discovery of available Grid resources that provide the optimum execution performance for a globally submitted tasks. The Grid, or global Grid, is a collection of multiple local Grid resources that are distributed geographically in a wide area. The act of balancing is achieved by discovering services. It is a indirect result which is effective across multiple Grid resources.
Agents use the processes discovering and advertising as the way of dispatching a task to the resource matching best its requirements. These processes are cooperating activities, which means that agents cooperate with each other in information exchange.

Agents capabilities are represented as local Grid resource. Capabilities are advertised throughout the hierarchy. Agents can hold different ACTs (Agent Capabilities Table). The ACT store information about local capabilities or global discovered resources. To maintain the tables the agent can use data-push and data-pull tactic. ACT are updated periodically or event-driven.

Discovering available services is done as follows: Within each agent, its own service provided by the local Grid resource is evaluated first. If the requirement can be met locally, the discovery ends successfully. Otherwise service information in the ACTs is evaluated and the request dispatched to the agent, which is able to provide the best resource match. If no service can meet the requirement, the request is submitted to the agent up in the hierarchy. When the head of the hierarchy is reached and the available service is still not found, the discovery terminates unsuccessfully. Inherently these processes leads to a coarse grained load balancing even though it is a bi-effect on service discovering.

There are many interesting metrics about performance of the overall system in the paper. The experimental design consists of 12 clusters/agents containing each 16 hosts/processors. The results shows various performance measurements. The results demonstrate that the genetic algorithm (GA) has better performance than the first-come-first-served algorithm.
The discovery mechanism also shows an efficient balancing performance when coupled together with the GA algorithm. Such an agent-based framework is scalable, flexible, and extensible for further enhancements.

5.4 Load Balancing with a Swarm of Ants

The novelty that brings [37] in is the "CAS" Complex Adaptive System. CAS is used to describe certain biological and social systems [38]. Özalp and his coauthors think of a basis for a new possible programming paradigm in P2P. In a CAS framework, a system consists of a large amount of autonomous computing units so called agents. The motivation in CAS is its ability to exhibit what is called emergent behavior: Individual agents can be understood very easily whereas the system as a whole is not easy to understand. The authors point out that agents can be based on much more complex patterns.

As an instance of a CAS, and actually the authors inspiration, they consider a colony of ants. Several colonies of ants are known to group objects in their environment (e.g. dead body corps) into piles to clean up their nests. The behavior is not coordinated by any ant. Agents can be seen as ants. The authors state what renders CAS particularly attractive from a P2P perspective is the fact that global properties like adaptation, self-organization and resilience are achieved without explicitly embedding them into the individual agents.

To pursue their work they have developed a P2P framework called Anthill [39]. It adopts the ant colony paradigm. In this agent-based approach, ants move across a network of nodes trying to achieve a particular task. While moving, they produce some product (output) and modify the task. The system may be defined as complex adaptive one, but individual ants are very simple. Single ants do not have a problem solving capability. Despite that, ant colonies manage to perform several complicated tasks.

The Anthill framework implementation is based on JXTA [40]. JXTA is a open source P2P platform proposed by Sun Microsystems. It is based on a set of open protocols and it is one of the most mature P2P platforms currently available (Java implementation). JXTA creates an overlay network which forms a hybrid topology. Some peers are rendezvous peers (similar to a super node in Kazaa [9]), gateway peers which deals with the NAT/Firewall problem, and the rest are normal peers. Rendezvous-peers form a interconnected network, maintaining indexes of their resources. Resources are peers holding some data which are published on the rendezvous peer (please find more about rendez-vous networks on [41]). Peers communicate through the overlay network. The benefits of basing the implementation on JXTA are several. For example, JXTA allows the use of different transport layers for communication, including TCP/IP and HTTP, and deals with issues related to firewalls and NAT.

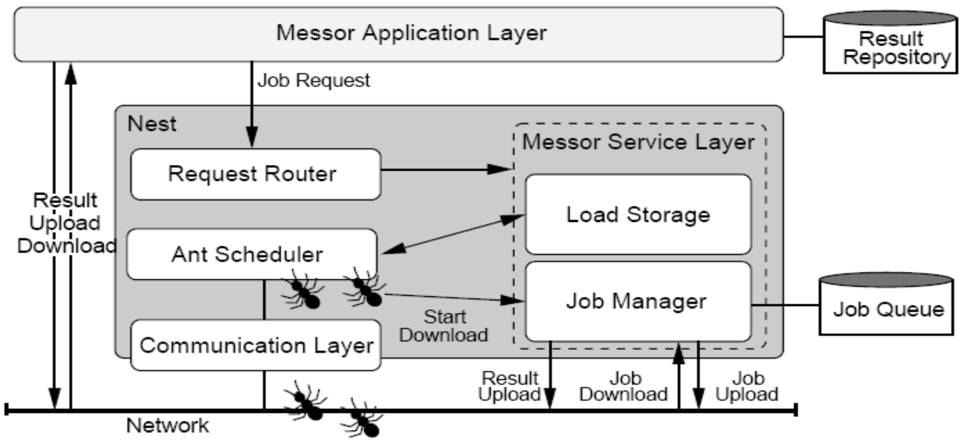


Figure 13: Messor Architecture

Messor is the name of the Anthill framework. Messor is aimed at supporting the concurrent execution of highly-parallel, time-intensive computations, in which the workload can be decomposed into a large number of independent jobs.

Messor is composed of a self organizing overlay-network of interconnected nests. Each nest is a middle-ware layer running on a computational node. The network is unstructured and loosely coupled: nests can come and go. Nests can communicate and discover each other on top of the communication substrate. The nest middle ware offer services to the running application on the node. Applications are the interface between the P2P network and the user. Services are implemented by means of ants: autonomous agents able to travel across the network. Ants are created in response of user requests.

The application layer, which is concerned with user interactions and collection of computed results and the Service Layer which is responsible for task execution and load balancing.
The service layer exploits the ant communication and scheduling facilities provided by nests. Load Storage contains information about estimated load of remote nests. The job manager is responsible for executing jobs which are assigned locally. Jobs are put on the

job queue after receiving a local job request through the Request Router or after downloading a job from a remote nest. Load is defined as the number of jobs currently in the job queue, or, if available, information about potential computing power. When ants wandering around, information of the visited nest is collected and put in the Load Storage of their home nest.

The system is resilient to failures, as jobs assigned to crashed nodes are re-inserted in the network by the nest that generated them. Messor is self organizing: New nests may join the network, and their computing power is rapidly exploited to carry on computations as soon as ants discover the nest and start to assign jobs from other nests.

The messor ant algorithm: Messor Ants live in a network of nests. During a ant's lifetime it can have two states: SearchMax and SearchMin. While in SearchMin state, the ant wanders around looking for an “under loaded” nest. When such a nest is found, the ant requests the local Job Manager to transfer jobs from the overloaded nest to the under loaded one, and then switches back to the SearchMax state again; then the process repeats.

The performance metrics are how the load balancing evolves over time steps. One time step consists of the ants running its run-method and moving to the next nest. A simulation with 100 nests and 10'000 initially created jobs shows that in 50 steps the load is evenly distributed over all nests.

5.5 Summary of Survey

Table 1 gives a summary and comparison between the studied papers.

The (1) has a clear benefit for its topology. All nodes are equal and topology maintenance is very simple. The network is highly adaptable and a complete ad-hoc network, as known for decentralized flooding models. But where ad-hoc benefits, the pay-off is the integrity of the network: it is clear that such network is not much use where deterministic large scale approaches must be achieved, since it is a unstructured network and highly unstable.

Another benefit from the underlying ad-hocity can be drawn, where it might exploit small world properties. The work stealing algorithm clusters active objects on high performance nodes, since nodes with more performance will also do more work.

The (2) uses a DHT as the underlying network. Since DHTs have be proved to be applicable in global large scale networks it can be applied for large scale networks. Because load is shifted by join/leaves in the structured network and occurrence of delay in the overlay due to communication latency can lead to inconsistency, harder constraints on overlay maintenance must be defined. In presence of node failure, there might be delays until a stable state is reached. Considering vulnerabilities, the overlay must provide security mechanisms to prevent threats.

Load balancing is taken out by some directories which schedule reassignments of regions (relocating the tasks). Because the design is adopted from a centralized scheme, it can easily be implemented.
A possible problem space is a large number of global dispersed nodes (Internet like), which can communicate over a reliable network, with good bandwidth respectively assured QoS of the physical network.

In (3) the process of discovering resources has a global resource balancing effect. In the hierarchical structure, each agent keeps ACTs (Agent Capability Table). When an agent learns about a new resources or it got an update, it keeps this information in such an ACT. Load balancing is therefore not actively ratter keeping the information in these tables upon change. Therefore the actual load balancing is a lookup, and route the task wherever it matched criteria. Note that in (1),(2), and (4) the act of load balancing are actively, and taken out immediately or in a certain period.

The interesting part in (3), and the only considered, is that the global load balancing is separated in two independent tasks:

- updating tables, and
- routing tasks.

The scheme inherently introduces a new layer, and enables complex designs. In this design routing algorithm could be implemented, such where other criteria can be taken into account: proximity, QoS, policies, brokering.

		Migration based LB (1)	***Balancing with structured P2P (2)***	***Grid LB using intelligent Agents (3)***	***Anthill (4)***
	Nodes	Computational nodes (JVMs).Load Balancing for Active Objects.	Data Items holding meta data: information like the memory size or processor time needed to serve a task. Nodes are also the computational entity.	Cluster of workstations or multiprocessor. Parallel virtual machine. Grid resources (represented by an agent). Local Grid resource scheduling and global Grid load balancing. Parallel task exec.	Nodes are nests. Jobs are distributed over nodes equally. Nodes are computational entities.
	Topology, Overlay Network	random graph, decentralized topology	ring, decentralized resource routing model	hierarchical, agent {Broker, Coordinator, Self}	Hybrid topology, resource routing model
	Builds on	ProActive middle-ware: active object programming model, group communication migrateTo(). ProActive uses Java RMI	Can be implemented on a overlay network such as CHORD.	Agent infrastructure, service discovery and advertisement P2P system (cooperative activity). These processes in local domain, among neighboring agents allowing scalability.	Anthill runtime environment based on JXTA. Self-organizing overlay network of interconnected nests.
	Load	Active Objects	Tasks	Tasks	Tasks
	balancing objective	Reduce the time for wait-for-necessity (WbN). Speeds up execution time of active objects.	Minimize load imbalance on the DHT by satisfying the requirement of minimizing the amount of the load moved (they are not orthogonal). Only load on virtual servers are taken into account (no prediction).	Agents use PACE evaluation engine to predict load on-the-fly. Input: Task and parallelism characteristics (taken into account the execution environment, HW and MPI/PVM application performance). Output: in the form of overall execution time estimates.	A nest's load is the length of the job queue or information about the potential computational power. Distribute load upon these info evenly overall nests.
	Algorithm approach	Sender initiated migration: Probing random subset of acquaintance nodes, migrates active object to a under loaded node (IFL). Receiver initiated migration: Work stealing approach clusters active objects on high performance	Using Virtual Servers: move load by reassigning the set of region associated to a node. Employs static scheduling idea in a distributed fashion: many-to-many scheme: periodical load balancing of all nodes, using directories. one-to-one: emergency load balancing for one particular overloaded	Using AI scheduling algorithms. Local: First-Come-First Served or Genetic Algorithm, Iterative heuristic Global: service discovery results in a load balancing across multiple Grid resources. Using X_ACT (Agent Capability Tables, where x is {self|local| higherl}).	Anthill: Derived from the Ant colony metaphor. SearchMax & SearchMin. SMAX: wander around until overloaded nest found. Switch to SMIN, wander until under loaded nest found. Ant then requests job transfer on local manager. Switch back to SMAX

		Migration based LB (1)	***Balancing with structured P2P (2)***	***Grid LB using intelligent Agents (3)***	***Anthill (4)***
		nodes.	node.		
	State abstraction	Rank criteria.		Predictive: estimates execution time.	Queue length, computational resources.
	Resource discovery/ dissemination	Query based mechanism, initiated by the resource	Resource dissemination, periodically pulls/pushes. Resource initiated	Resource discovery, agent based. MW initiated initiated.	Match maker: m/w initiated
	Performance measure	Measured in #of migrations and the IFL performance: ratio between #of nodes used by an optimal statical distribution and #nodes used by the IFL algorithm.	Load movement factor in function of system utilization. 99.9th percentile node utilization in function of load movement factor	Total application execution time. Average advance time of application execution completion. Average resource utilization rate Load balancing level.	Load Distribution in function of iterations. A iteration corresponds to a set of ants running their "run method" and moving to an other nest.
	Analysis, scale	Simulation of the algorithm on a large-scale P2P Network (till 8000 nodes). Simulation pattern adapted from real world: seti@home cpu-usage distribution.	Fixed 4096 nodes, 12 virtual servers per node and 16 directories, average number of objects: 1 million. Different Patterns: Non-uniform object arrival patterns, node arrival patterns (Poison process)	Experimental system with 12 agents. Experiment 1 employs first come first served result poor cluster utilization. Experiment 2 employs the iterative heuristic one results in good improvement	100 Nodes with initially 10'000 Jobs created in one node.

Table 1: Characteristics of surveyed load balancing solutions

Every agent has a different view of the system (local tables). The pro is that information is available at place. On a network wide view, each table differs because each node has a different view of the network at any time. Anyway, the interesting aspect here is that the manipulation of the ACT table happens at discovery time, compared to where it is achieved when lookup for the best suited resource is done.
(4) is a match-making ant wandering around and being diligent. An ant is wandering from nest to nest, and if a overloaded nest is found it searches for an under loaded one. When matching condition is met, tasks are transferred. Having wandering ants implies having a reasonable overlay network enabling it. This is achieved in (4) through JXTA.
Different questions arrives in the self management of the ants. When is a system to be regarded as overloaded? There might be different reasons for having many visits of ants, such as too much load in the system and network partitioning or creation of ants upon unreliable information.
The interesting part here is that the balancing logic is implemented in the ants. This approach allows as in (3) more complex balancing mechanisms. What it makes clear different to (3) is the act of balancing which actively happening compared to (2) where it is passively happening. Nests offer mechanisms to move load from one nest to the other. The ant can therefore invoke these mechanisms.

6 Design

The motivation in the design is at the heart of request routing. Request routing has the property that it can be divided into two independent processes: The gathering of routing information process and the routing of request process based on routing information.
First I introduce the concept of request routing. The model of the system will bind us then to our problem space as described in the second part of this chapter.

6.1 Concept of Request Routing

In this work, a request is defined as the following: A request is a process where a consumer asks the resource provider for execution of the requested Web Service and may receive a response which can be a request-response or a request-error.
The request-response is the outcome of invoking the service successfully or unsuccessfully. A request-error is the response of the resource provider, unable to invoke the service (e.g. when there is no way to invoke the service).
The consumer is the potential entity willing to use resources of the Grid. In this work, a client or service-consumer is referred as the entity producing requests. The Grid on the other hand is the service-provider.

.

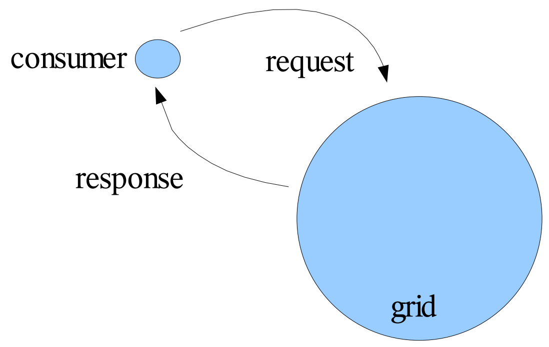


Figure 14: Consumer Request

Note that in Figure 14 the consumer could also be a node in the the Grid. In the context of GRAM (GT) a job request is a request to gatekeeper to create one or more job processes, expressed in the Resource Specification Language. This request guides

- resource selection (when and where to create the job processes);
- job process creation (what job processes to create);
- job control (how the processes should execute;

Clarification of terms which often occur in this section: a *logic* refers to an algorithm, a program code or just some code that has a significant importance for the overall functionality in context of its occurrence.

6.1.1 Life cycle of a request

A request passes 3 steps in its life:

1. issuing a request: life begins with a certain aim;
2. routing a request: life becomes to satisfy the aim;
3. acceptance of a request: aim found, life ends.

By analyzing the 3 steps I will bring the context up to make the concept and requirements.

6.1.2 Creating and issuing a request

Creating a „fully qualified“ request is built up on two information

- request constructing: knowledge to build the request data structure, and
- protocol entry point.

A potential consumer needs to consume a service. Therefore the consumer knows why and what it wants to consume. However, we are not concerned about why and what the consumer wants to consume; we are concerned in how to consume the service. For example in Web Services the consumer will get first an WSDL document. This document is partially like a blueprint for producing customized request for services. In this case, the consumer has the recipe to produce the needed request structure. This example explains that there must be some knowledge about how to build a request.

The protocol entry point is needed to specify where to start with that request. It's rather a theoretical meaning (specification) but necessary for a request to be useful. The request routing, explained in the next paragraph, can be entity based (consumer centric), remotely based (resource centric) or in a mixed fashion depending on the specification. This simply states how request routing will be taken out. Because a request is a data structure, without any entry point the request will not do much. The protocol entry point is inherently needed. By knowing the protocol entry point, a request can be given to the protocol and issued.

6.1.3 Request Routing

Routing is the catch all term for finding the way towards a specific destination with help of an algorithm called routing algorithm. An employed routing algorithm defines fir which objective finding the path will be made e.g. on path cost and others but it also can be an algorithm which aims in load balancing. The algorithm needs some input called routing information. The output produced is the routing decision.
Routing information is the information up on which the routing algorithm produces the routing decision. Routing information usually maps a route (next hop) in respect to a objective (like a cost objective). Routing decision is the decision containing where the subject (e.g. a request) will be addressed to in the system. It's produced by the routing algorithm.

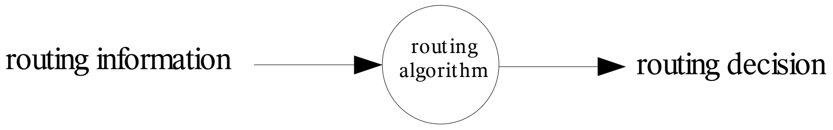


Figure 15: components of request routing

This mainly defines where the logic of decision making sits and what interfaces consumer and provider must offer each other.
In the source centric request routing, it is up to the consumer to retrieve information, process information and make the decision where to submit the request. In this case the logic and responsibility is in the hand of the consumer.

Source centric is also useful when interaction of a human or another instance is needed. Consider the case where a consumer first has to contact a broker: A consumer asks a broker for some resources to be used by passing a specification. The broker in turn delivers an offer with different resources for different costs. The consumer, or human operator, chooses then which resources fits best.

In the resource centric request routing, it is the resources which take care of the routing integrally (like in IP routing). The logic is based in the system. That is easy for the client, since it only has to submit the request and the rest will be done by the system. One could argue that a broker could be include too here which receives in-line with the request some „money“ or credit for which it should find accurate resources for. Of course that's possible but it shows that choosing a different design approach designates a different architectural aspect, and vice versa.

6.1.4 Request Routing Types

I divide basically request routing in 2 types , Figure 16, which is either a passive lookup type or an active lookup type.

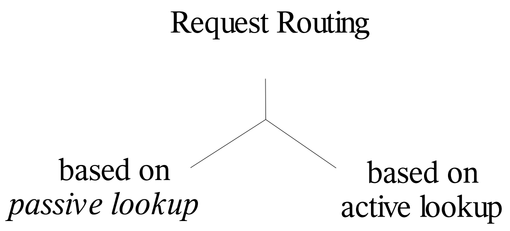


Figure 16: Request Routing types

The main difference between active lookup and passive lookup is how and where the information for routing decision is obtained. The information feeds the routing algorithm with the necessary information such it can make a decision.

In the passive lookup, information is available locally meaning that the routing decision can be made at the place with the given information. On the other hand, active lookup first collects and processes information and then makes routing decisions. The benefit of using active lookup is to taking into account the current system state or system topology. A trade-off for the active lookup is the delay before the outcome of the routing decision.

In the case of passive lookup, decision can be made immediately and independent and no delay occurs. The trade-off here is the system state might be outdated, and also topology might have been changed and therefore not adaptive to dynamism (think of failures, join and leaves). To deal with dynamism, an update must be achieved *periodically* or on *demand.*

In the passive type we need some logic which runs to update our local routing information. This implies that during the update process, messages within the Grid are send and received, memory and processor cycles are used. This happens maybe on a periodical or trigger based scheme. Anyway, the characteristic here is that there's a constant update process running consuming resources (overhead).

In the active type the logic to prepare the routing information is triggered when needed, messages are send and memory and processor cycles are used. This implies that if there's no request to be routed, there will also be no overhead.

In a hybrid type of lookup, one could imagine that each node in the Grid holds information with a certain *durability*. When the information out dates it will first be updated. An update process might push also information to other nodes. If a routing has to be performed it will first check if the local info is out of date or not and takes the necessary steps.

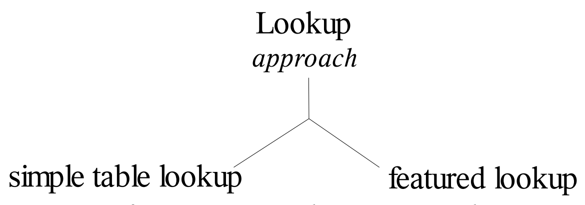


Figure 17: Information Lookup approaches

The information lookup is the process of feeding the routing algorithm with the necessary information such it can make a decision.
The most simple one is the simple table lookup (figure 17). The table contains the information and the decision can be made with a very simple decision logic. The featured lookup holds a bunch of information which can be treated with a more complex logic, resulting in many operations and intermediate results. The featured one is clearly more extensive than the simple but also more powerful. When the featured is employed as an active lookup type, the lookup logic might become an essential part of the routing algorithm itself.

In the passive lookup type, the information is stored locally. Anyhow, information must be made available first. Two schemes categorize availability of *centralised* and *distributed* information.

The *centralised scheme* make use of a central logic where information can be pushed to or pulled from it. An implementation for example could be a central directory. Nodes can push information into it an get information out of it. The directory might also broadcast information to all nodes or subsets of nodes.

In the *distributed scheme*, information is gathered from many nodes at many nodes. There is no central known directory. A good example is aggregation of information.
For a more specific classification of a distributed scheme, I divide the distributed scheme into two subtypes where the underlying overlay dictates the fundamental interfaces for communicating among nodes:

- random topology (we do not care much about overall network boundary), and
- deterministic topology (network boundaries can be exploited).

In the latter case a DHT constructs a *deterministic* topology. The lookup mechanisms can be exploited. The main point for deterministic topologies is, tha boundaries of the network can be estimated, where in unknown network only a certain part for each node in the network is recognised. Nodes have different views of the network. Where the underlying overlay is a random topology, mechanisms like neighbour querying, information dissemination, network flooding or other mechanisms must be used to gather information.

6.1.5 Accepting a request

There are two possibilities of what the resource can do with a request: the request can either be executed or not. In the latter case the result will be a rejection for various reasons. The resource can, but must not, create a response to notify the involved entities. When resource has executed the requested wish, it creates a notification with either a positive or a negative result. The positive result is the natural outcome of the computation. The negative result states that a computational error occurred. In the case of a fault in the resource or system, a response might never been created, received or sent.

6.1.6 Balancing requests is balancing load

Recall figure 15 (components of request routing). Figure 18 demonstrates where load balancing could take part.

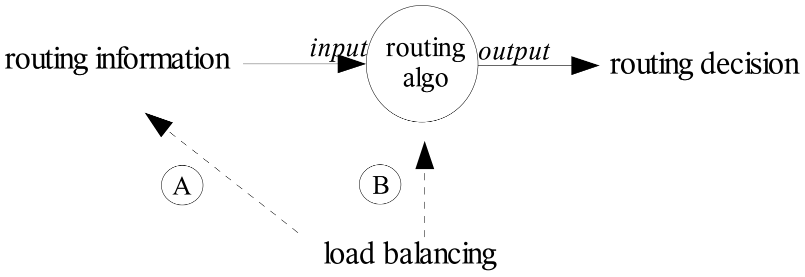


Figure 18: request balancing enforcement points

In (A) the balancing of request is based on the current routing information. For example a distributed algorithm runs in a system periodically to create a list with nodes ordered by their load. A lower load reflects a lower cost. The routing takes the list as input. The routing algorithm would therefore make a decision considering the route to a node with the lowest load. Enforcing routing based on the routing information is a passive lookup type.

On the other hand, in the case of the active lookup type, the routing algorithm itself is the load balancing. This means that the creation of the input is part of the routing algorithm itself. This involves information acquisition together with other nodes.

6.2 System Model

The problem space is given by the Grid4all project, which aims to enable non-profit users such as schools and private people and small enterprises to share resources to harvest massive Grid resources [42]. To achieve such a system, Grid4All focuses on a system which is composed of structured P2P overlay services and self-managing and self-organizing (being a self-*). It should allow to scale to a large number in the Internet and be highly dynamic to create ad-hoc organizations where participants can join and leave upon their own interest and need. The lifetime of a organizational coalition might be between a few hours up to multiple years.

Within this given system properties a suitable solution for introducing load balancing will be elaborated.

6.2.1 The system model

GRAM4 components are used for job execution and control on local resources (node or the physical machine). A routing algorithm executed by a distributed meta scheduler is responsible for routing a job request to a resource. The *routing algorithm* enforces the load balancing of the overall system. The routing algorithm is distributed, and every resource in the Grid runs the same algorithm. The resources can communicate with each other without keeping a local index of all nodes. There might/might not be need for the capability of communicating to all nodes but at least to a subset of nodes.

A node is a computational resource with the ability to do routing. Routing itself is taken out by the *routing-component*. The resource is the component which offers computational power to the Grid. A resource is e.g. a machine, PC, desktop computer and so on, running the Grid middle ware. A node is therefore built up of a routing component and the middle ware enabling its resources being used in a Grid VO.

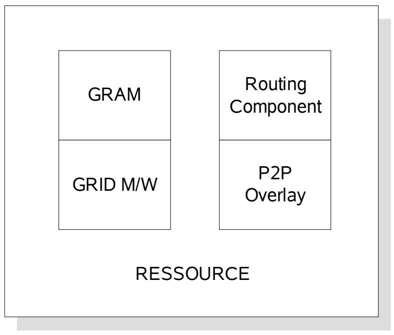


Figure 19: Components of a node

Grid-Job in a VO

A client submits a job request and runs the routing algorithm. The outcome of the algorithm will be one out of three decisions: either it accepts the request and executes it locally or it routes the request to an other node or or it will reject for a known reason by informing the client

The node which decided to run the job must have a resource which fulfils the criteria to execute the job given for example by a specification. If the resource is not able to execute the job, it rejects the request. Therefore routing concerns itself only with the distribution of jobs.

GRAM-job

Once a node has accepted a request and passed it to GRAM, it follows the GRAM scheduling model. The user which submitted the request will be in direct communication through the GRAM. Every node has a view on the overall system load. This might be a individual view and each view of the resources might be different. If the overall system is saturated, the system should reject any new request by informing the client (graceful degradation).

The system must be capable to handle system dynamism. Since real systems will have resources which joins, leaves or fails system dynamism is a crucial issue which must be dealt with. Nodes might not have the same view of parts of the system.

When it comes to scale, the system should be able to scale 10s of thousands of nodes. In other words, the system should be very scalable.

When it comes to saturation of the utilized resources, the system should gracefully degrade and reject new requests. Some nodes in the system might detect saturation albeit some do not.

The metric on which load balancing should be based on the queue size and capacity a node provides at any given time. Based on these two metrics different objectives of load balancing can be faced such as routing a request to a node with high capacity and/or with low utilization.

Requirements to the resource
The resource must abstract the local capabilities to provide the needed metrics These are a measure of queue length and overall execution time of the running jobs. Each resource has a GRAM component which controls and runs the jobs locally according to the the GRAM scheduling model.

Requirements to the client
The client is a node in the system. It submits a job request by submitting it locally to the routing component. Whether the request is accepted/rejected locally or remotely should appear to the client transparently.

Requirements to the requests
A request is based on the GRAM architecture. It might contain also some specification which is understood by GRAM. Anyhow, in this work we are not concerned with any job specification.

Requirements to the communication substrate
As the communication substrate between the resources, a P2P overlay should be used. The most important dependability property to the overlay is 'dynamism'. Therefore the communication substrate should reflect that property which allows to deal with failures, joins and leaves in the communication network fulfilling the self management requirement.

The QoS which the Grid offers is regulated in the participant policies.

Based on this system model and according to the request routing model we will employ a load balancing mechanism with the following properties:

- *use a passive lookup type,* information must be available when requests arrive
- *use a featured lookup* to gather routing information
- Information is distributed on a deterministic topology (DHT)
- load balancing decision is made on the routing information

Therefore we have two main mechanisms for the load balancer

1. algorithms of gathering routing information
2. algorithms which routes request based on results of point 1

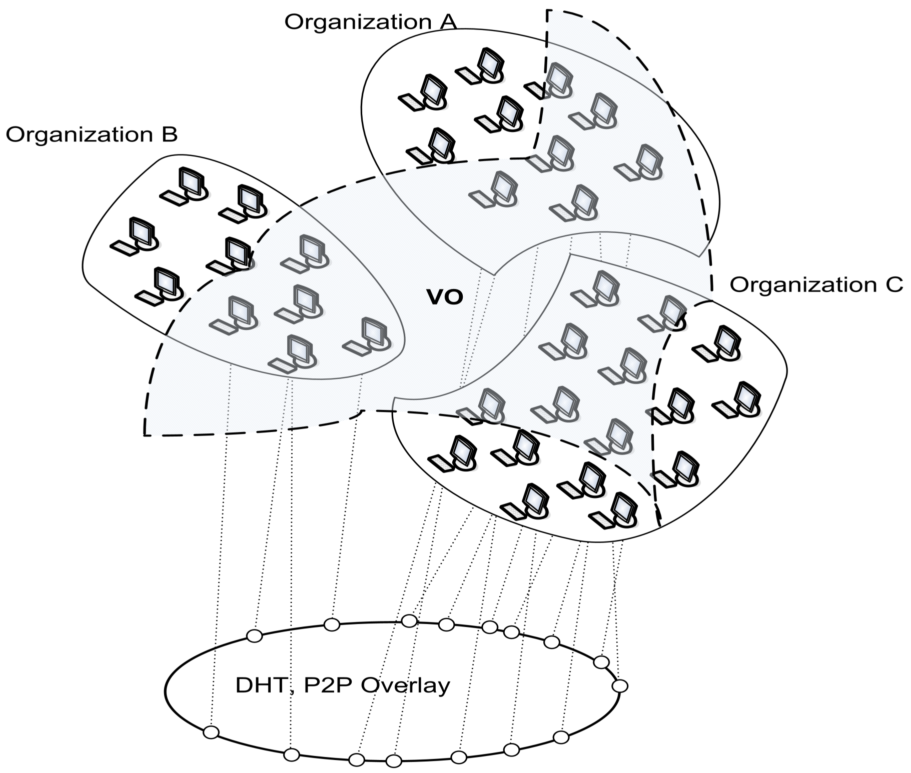


Figure 20: VO using a DHT overlay as the communication substrate.

We limit in this thesis the definition and evaluation on the first component, which deals with the *gathering of information*. The second part is given in short as a vision of possibilities to actually implement load balancing in request routing.

6.2.2 Inspiration from the survey

From the anthill paper [39]we could borrow the idea of the ant which wanders around together with the idea from the paper [34] to keep a table with capabilities of nodes. In our case the ant would inform higher loaded nodes where it has seen lower loaded nodes; the nodes could keep a table with that knowledge, and route an incoming request according to that table.

From the first paper [32] we could use the way how nodes measure the load from their neighbours and accordingly build a table during discovery. Combined with a flooding message protocol, where nodes can learn and update their capability tables of inspected nodes. Since we are not dealing with job migration, stealing work from higher loaded nodes is not treated and not part of the routing. However, it could be seen as a supplementary feature to the existing balancing system.

The survey gave us a facet of different solutions for load balancing for different problem spaces. We learned what technologies are used for what type of problem environment. Anyhow, we will focus on a solution based on a structured overlay network with a full decentralized architecture, where the survey gives a fall-back if necessary.

7 The structured aggregation scheme

We will describe the information gathering process in this part as the main contribution to the thesis work. We describe the idea of scalable n-aggregation in large scale systems.

7.1 Introduction

Aggregation is usually referred as the process of composing a multitude of information into a single one. An example of distributed information aggregation is the Grid resource monitoring in Globus MDS [43] where continuous aggregation tells on the monitoring entity the CPU usage of Grid resources.
In our case, we are interested that the aggregated information is omnipresent, and not only at a single place. Omnipresent means that everybody has a view of the aggregated information in a system, and therefore called ***n*-aggregation** (n stands for n participants). Thus everybody is monitoring. Therefore we use the term "aggregation" only to describe the way of composing the information. We are considering a continuous aggregation.

Multiple aggregation schemes have been proposed to leverage the topology information of structured P2P networks [44]. Anyway, in our work we focus on aggregate information in such a way that it is available on *every node* in the system, and not only on a single node.

Gossip-based aggregation [45] is as far as we know the most promising, since it was shown to behave robustly in large dynamic networks and also speaks greatly for its simplicity. The Jelasity paper's [45] gossip protocol inherently makes all the aggregated values available to each participant in the protocol.
The main difficulty for [45] is that the set of nodes known to a node must be random enough, and randomness must be maintained. The paper doesn't give any real-world practical evaluation for their protocol under churn, which we will evaluate.
The algorithm does not have information completeness, which is based on the assumption that the aggregate is calculated based on the knowledge of a subset of values [46]. In the paper [46], Gupta et Al propose an abstract hierarchical scheme with a gossiping protocol to aggregate in a system of large process groups. Aggregations are built on subsets.

Our scheme is inspired by those two papers [44] and [45]. Building the knowledge on subsets will allow us to use each subsets information for later routing purposes, and it allows a practical scalable omnipresent aggregation.

We will show a scheme where load is aggregated. We will not discuss the actual load balancing act since it must be treated in a own project. We concentrate on the issue of gathering information, such that load balancer can utilize the information.

7.1.1 System model

The model is a distributed system, built up of nodes which communicate through message passing. The system is asynchronous and therefore the upper bound of time needed to deliver a message is not known. The communication channels are not perfect, and messages might be lost. During our evaluation we do not consider message lost. Failures of nodes might occur as it does in real world. Failures represent the absence of a node which was assumed being present.

7.1.2 Algorithmic Notations

We use the event driven notation. Nodes receive messages and the messages results in an event. An event is a procedure, which also can send messages to nodes. An event can be initiated locally as well.

7.2 The Structured Aggregation Scheme

We employ a Distributed Hash Table. The DHT is split into different intervals, where intervals have different amount of members depending on the interval size. A node belongs always to a predefined interval. Such that nodes collect information from other nodes they communicate over the DHT. Nodes have a predefined scheme with who they are allowed to communicate to. This is given by the interval they reside in. Each member of an interval is allowed to talk to members in its complementary interval.

7.2.1 Underlying Structured Overlay

The algorithms are based on the underlying overlay network, which is a structured network. The overlay is CHORD (which reflects the general case)[4]. Note that all arithmetic operations will be modulo arithmetic to the size of the ring (N). When we talk about a node with identifier *i* we use from here on the shorthand node *i* for it.
CHORD can be modeled as a graph $G=(V,E)$ with $n=|V|$ nodes and the edges E are the connections on the overlay between the nodes. A node $v \in V$ is denoted by ID(v) which is a unique identifier in a *b*-bit identifier space, where $ID(v) \in [0,2^b)$. Chord assigns identifiers to nodes using a consistent hashing scheme.
All nodes organizes themselves in a ring topology where the identifier ID is the position in the circular space. Each node n has a predecessor denoted by *n.pred* which is its immediate predecessor node. Each node has also a successor denoted by *n.succ* which is its immediate successor node. The direction in the ring is clockwise.
The overlay has an identifier space consisting of size N, $N=2^b$ which defines the maximum number of nodes. Chord uses fingers as shortcuts for lookup. The fingers from a node *n* point to a node such that the first finger points to *n+1*, the second one to *n+2*, the third one to the *n+4* one and so on. There are $f=b$ fingers so $N=2^{|f|}$ Each finger *i* at a node n points to a node

$$f_i^n = n \oplus 2^{i-1}, 1 \leq i \leq |f|$$

The principle is going from the highest finger down to the first one in steps. In each step the identifier space is halved. By using the fingers, routing can be achieved in logarithmic complexity.

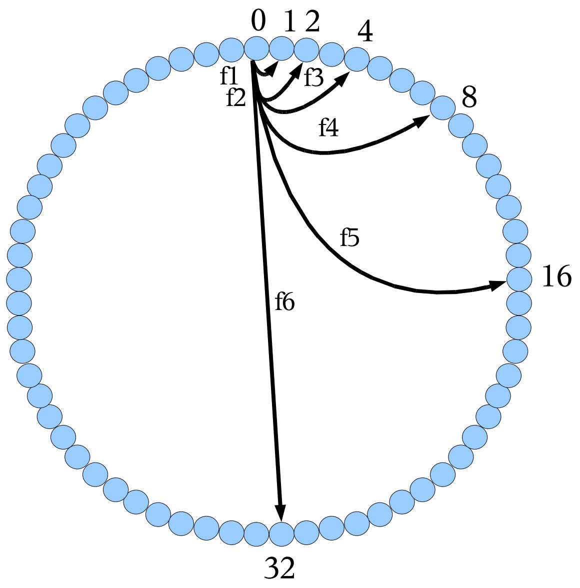


Figure 21: finger pointers for node 0 in a ring with N=64

7.2.2 Structuring the gossip

The basic aggregation protocol from Jelasity et. Al [45] uses a push-pull gossiping scheme. The protocol has two threads, where the first is initiating a state exchange with a random node and the latter thread is passively awaiting for state exchange. In one cycle nodes exchange their states, and after one cycle both nodes holding the same weighted average (the estimate). In each consecutive cycle the variance over the set of all estimates in the system decreases. It was shown that after a little number of cycles the variance over the set of all estimates converges near to 0, which means there exists a fast convergence of the protocol.

The inspiration of gossiping in our work is the following: Regarding the ring structure, we could easy implement the gossip protocol [45]. Sending a message to a random (destination) node is on CHORD routed to the closest proceeding node of the destination node. Whoever this closest proceeding node is, it will be routed to that node. Random choice can be achieved by choosing an identifier i from the identifier space. To give good randomness we assume N is big.

Since we structure gossiping into a structured scheme, the term gossiping doesn't really match its purpose anymore. We simply call it (restricted random) information exchange.

In our scheme, random information exchange is processed in different levels. Lets take a DHT, with an identifier space of size N. We introduce levels, such that the information can be composed of these levels. Each level allows the node to exchange information within a certain hierarchical scope. If a node n exchanges information with another node so called opponent, then the opponent gives the information as well to node n.

Interval levels

An interval is a sequence of consecutive IDs on the identifier space. Levels define the size of the intervals.

To build a *composed* aggregation, information exchange on the ring is structured into a defined protocol. We construct so called levels which contains intervals and each node will know exactly with which *opponent interval* it is allowed to exchange information for which level. Each node will belong to an interval in each level and for each node exists an opponent interval in each level.

In one cycle, *N* nodes will exchange information for one level. Since there are $L=\log_2 N$ levels there are L cycles.

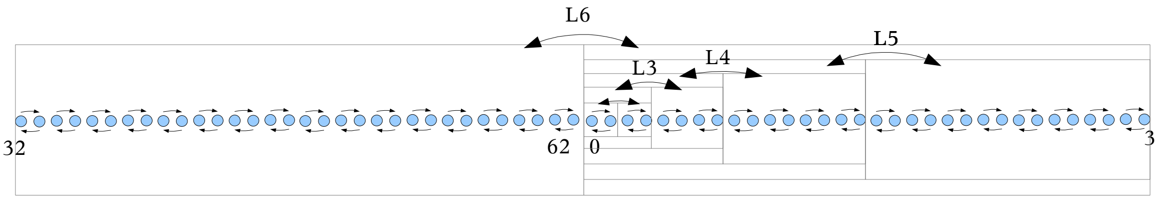


Figure 22: Interval levels (shown only in the right half), in a ring of N=64

On top of the DHT we create intervals *I*. Each defined interval belongs to a level *l* in the DHT. A level defines the size of an interval, where the size of the level is 2^{l-1} nodes.

$$I_l^t=[k\cdot t, k\cdot t\oplus k), k=2^{l-1}$$
$$t=\{0,\ldots,\frac{N}{k}-1\}$$
$$l=\{1,\ldots,\log_2 N\}$$

A level l has $\frac{N}{2^{l-1}}-1$ intervals. Therefore each defined Interval I_l^t is the *t-th* interval in level *l* . Formally, an interval level is I_l^t where t is the numeration of intervals, always starting at *t*=0 for each level *l*. Each node can compute according to its ID to which interval and opponent interval for each level it belongs to.

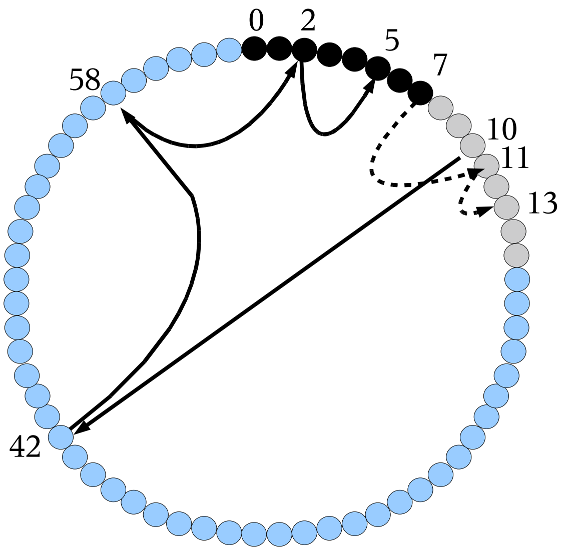
The following rule will guarantee that nodes, in each level, will always communicate with a node from the defined opponent interval and vice versa:

$$\text{node n in level l belongs to interval } t: n\in I_l^t$$
$$\text{opponent interval } =\left\{I_l^{t'} | t'=\begin{cases} t-1 & \textit{if odd}(t) \\ t+1 & \textit{if even}(t)\end{cases}\right\}$$

For a node *n* in interval *t* of a level *l,* the opponent interval of *n* in level *l* is $I_l^{t'}$.

Nodes in an even interval t will always choose nodes in interval *t+1*, and nodes in the interval *t+1* will always choose nodes from interval *t*, and therefore nodes from *t* and *t+1* will only communicate with each other. A node chooses an opponent randomly (*with uniform distribution*) from the opponent interval.

Asymmetric cost on the overlay: Since messages will send towards 2 mutually intervals where the intervals are constructed in clockwise direction, there exists an asymmetric cost. Because nodes in an odd interval, say node 5, will send message in counter clockwise to interval 4 but the construction of the CHORD finger table is in clockwise direction. There are significantly more hops since finding successor in CHORD is achieved in clockwise direction and the lookup will go around the clock.



Drawing 1: 2 intervals: upper interval [0,8) has lower cost (dashed line) in hops, where lower interval [8,16) has higher cost. N=64, chosen level=4.

In the results of the evaluation, we will discuss the measured costs for our scheme. Note, for an exchange of information between node *n* and *m*, where *n* is the initiator, only node *n* looks up node *m*. Node *m* will reply directly to node *n*, resulting in only one single hop.

7.2.3 Data Structures used in the Algorithms

The following notations are used for data and data-structures in the aggregation algorithms.

lm: "load message", is a data structure holding the information used during aggregation. Each message sent is a lm data structure.

rL: "received Levels", is a list of messages (type lm). The index of the list is the level the message corresponds to. A node stores the information (load messages "lm") received in rL.

cL: "computed Levels", is a list of messages (lm). The index of the list is the level the message belongs to. The cL is constructed respectively calculated from the rL list. A cL has L+1 values, where L is the number of levels:

generally cL is: $cL[1 \ldots L+1]$
$cL[1]=$local load
$cL[L]=$ load estimate representing half of the ring
$cL[L+1]=$load estimate representing the whole ring

Procedure "lm:=OP(loadmessage,loadmessage)": is a placeholder for the implementation of the aggregation calculation. OP contains the objective function how data should be aggregated (average, mean, count...) .

L: "The number of levels": $L=\log_2 N$

I $_j$: "Interval of level j": The interval of *opponent* nodes for level j. An interval is always represented as [s,e), where *s* is the starting node included and *e* the ending node excluded of the interval. Each node knows its L-intervals and opponent intervals.

When the aggregation run has terminated, every node will hold a value for each level, where the highest level represents the global value. The value is subject to what the aggregation objective is, such as average, count and so on.

7.2.4 Simple scheme: Symmetric Scheme

Algorithm 1 initiates an information exchange. For each level, starting at level 1, a node *n* will send a message to a node *m* in its opponent interval. When *m* receives the message it will trigger the algorithm 2, and update its local information and sends it's current information about the current level to node *n*.

Node *n* either receives the response message from the node it previously sent the message or it doesn't. If there was no such node *m*, or if one of the messages got lost, node *n* will never receive a R-LOADMESSAGE. This is broken by a timeout, such that the algorithm can terminate.

If a timeout occurred, node *n* will set current opponent level to unknown, assuming that no node exists in that interval. If there exists some node in the opponent interval, then there will be a chance that a node of the opponent interval will contact node *n*. Every node in the system has to run algorithm 1 since nodes don't know beforehand if opponent level nodes exist or not.

Algorithm 3 updates the estimates. It triggers whenever algorithm 1 has run through or if a load message was received and matched the current level.

No synchronization of cycles: In the symmetric scheme there must not (but can) be a synchronization. Nodes do not have to proceed the protocol for each level simultaneously (as in [45]). Because we build the information based on defined logical groups on the ring, it will always be a weighted information based on the estimated size the level. This property decouples cycles from the obligation being synchronized and makes the protocol practical for real world asynchronous systems.

In a fully populated ring, an aggregation will always end such that every node holds the same aggregation value. This is true since every node initiates an exchange for each level, and each contacted node will reply and hence each level agrees on the same value.

If the ring is not fully populated, then an aggregation end such that every node eventually holds the same value. Due to absence of nodes in intervals, there exists also missing information exchanges and hence different intervals for same levels will have different values.

Algorithm 1 active information exchange, symmetric/asymmetric scheme, run periodically

```
1:  procedure n.EXCHANGE()
2:      for j:=1 upto L do
3:          m:= random r , $r \in I_j$
4:          sendto m.LOADMESSAGE(cL[j])
5:          receive R-LOADMESSAGE(f) from p,
                where $p \in I_j \wedge f.level = j$ or timeout
6:          if $timeout \neq True$ then
7:              n.UPDATE(f,j)
6:          else
9:              n.UPDATE(NULL,j)          //level is unknown
10:         end if
11:     end for
12: end procedure
```

Algorithm 2 receive load message, symmetric scheme

```
1:  event n.LOADMESSAGE(f) from m
2:      j:=f.level
3:      if $m \in I_j$ then
4:          n.UPDATE(f,j)
5:          sendto m.R-LOADMESSAGE(cL[j])
6:      end if
7:  end event
```

Algorithm 3 updating the estimates

```
1: procedure n.UPDATE(f,level)
2:     rL[level]:=f.value
3:     for k:=level upto L do
4:         n.OP(rL[k],cL[k])
5:     end for
6: end procedure
```

7.2.5 Simple Asymmetric Scheme

The difference to the symmetric scheme is nodes only send messages to a random node in its opponent interval in each cycle and never reply if a load message was received. Therefore the protocol can proceed after each sent message. The number of messages used is half the size as in the symmetric scheme, and thus half the message complexity of the symmetric scheme. The better message complexity comes inherently at cost of the accuracy of the calculated aggregate. For more details read the evaluation section.

Algorithm 1.2 and 2.2 are the modified for the symmetric scheme. In contrast to the original symmetric, it will never response on a received load message.

The algorithm 1.2 has a simpler protocol than 1, but needs some remarks. At each run of algorithm 1.2 the received level data structure (rL) must be zeroed. Since we do not have a reply mechanism where we explicitly determine if an interval has members or not, we must assume from beginning that there are no nodes in the interval. The initial value is then overwritten if load message in algorithm 2.2 was received.
According to algorithm 1.2 all nodes must not (but can) proceed synchronously. The reason are the same as explained in the symmetric algorithm.

In a fully populated ring the aggregation ends and eventually every node holds the same value. Since the protocol only sends information to a random opponent, we do not guarantee that intervals of same levels have same values since there's no bi-directional exchange.
If the ring is not fully populated, such that every node eventually holds the same value at the end of the distributed algorithm, the probability decreases.

Instead of resetting every rL and every cL at the beginning of the distributed algorithm, the algorithm could reset a level only if needed. This improves the algorithme, since it might not yet have received the load-message for the level it is about to send a load-message.
Anyhow, we would also like to assume that a level doesn't exist, since this is the default behavior. This can therefore be achieved by using a flag indicating a level was recently updated (within a duration of a full aggregation). The rL and cL will be reseted if for the current level no update rL has been received.
A flag could be set by algorithm 2.2 such that algorithm 2.1 can verify that there was an update for that value since the last full aggregation. The Algorithm would therefore reset the flag when it was set due to an update (each level has a flag).

Algorithm 1.2 active information exchange, asymmetric scheme, run periodically

```
1:  procedure n.EXCHANGE()
2:      for j:=1 upto L do
3:          rL[j]:=NULL
4:      end for
5:      reset cL
6:      for j:=1 upto L do
7:          m:= random r , $r \in I_j$
8:          sendto m.LOADMESSAGE(cL[j])
9:      end for
10: end procedure
```

Algorithm 2.2 receiving a load message, asymmetric scheme

```
1: event n.LOADMESSAGE(f) from m
2:     j:=f.level
3:     if $m \in I_j$ then
4:         n.UPDATE(f,j)
5:     end if
6: end event
```

7.2.6 Improvement of the asymmetric/symmetric scheme

Since node *n* chooses an opponent node *m* at random from the identifier space, *n* does not know beforehand if *m* exists. We assume nodes are uniformly distributed. Thus finding a node on the ring has in each interval in the same level the same probability.

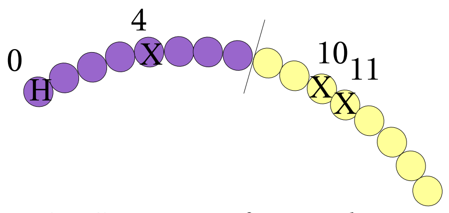
If an opponent interval is fully populated, then each random choice will be a hit. If the opponent node is not fully populated, then the probability of getting a hit is smaller than 1. Under the assumption of no message loss, each non-hit can not be distinguish between an empty or non empty interval. The symmetric protocol doesn't distinguish between empty interval or a missed node. It always assumes that the interval is empty. The asymmetric scheme simply doesn't care, and always set it to empty before each new round.

Such that more accurate aggregation can be made we propose a simple modification to the Algorithm 1 and 2.

Algorithm 2.2.2 improved receive load message, asymmetric scheme

```
1:  event n.LOADMESSAGE(f) from m
2:      j:=f.level
3:      k:=f.nodeID
4:      if $k \in I_j$ then
5:          n.UPDATE(f,j)
6:      else
7:          if f.resend = False
8:              f.resend := True
9:              sendto $H_j$ .LOADMESSAGE(f)
10:         end if
11:     end if
12: end event
```

If a node m receives a message, and m is not in the targeted interval of that message, then m sends the message to the beginning of the targeted interval. According to the CHORD protocol, the node receiving the re-sent message will be a node in the interval if the interval is not empty, or it will be received by a node outside the interval if empty.



Drawing 2: Scenario of resending message. X means node exists. H stands for "head" of Interval in clockwise direction.

Assume that in drawing 2 node 11 sends a message to node 5 in the opponent interval. Since node 5 does not exist, the message will be routed to node 10 according to the CHORD protocol of "find successor". Therefore node 10 would resend the message to node 0, which is the head of the upper interval. Since node 0 does not exists, the message will be routed to node 4. Node 4 accepts the message.

In the following modified algorithm we guarantee that in each level will be an information exchange, meaning that if opponent interval is not empty p=1 for getting a hit, and if interval is empty p=0 for getting a hit. Thus we achieve always a representative weighted aggregation in the whole system.

Remark to algorithm 2.1.2 and 2.2.2: The field 'resend' is in every new message initially set to 'false'. H_j denotes the first node (Head) in interval j for node n in clockwise direction.

Algorithm 2.1.2 improved receive load message, symmetric scheme

```
 1:  event n.LOADMESSAGE(f) from m
 2:      j:=f.level
 3:      k:=f.nodeID
 4:      if $k \in I_j$ then
 5:          n.UPDATE(f,j)
 6:          sendto k.R-LOADMESSAGE(cL[j])
 7:      else
 8:          if f.resend = False
 9:              f.resend := True
10:              sendto $H_j$ .LOADMESSAGE(f)
11:          end if
12:      end if
13:  end event
```

8 Evaluation

We will evaluate the symmetric and asymmetric scheme, compare them to each other and to the Jelasity scheme [45]. The evaluation is made within a specially developed Java simulator. The simulator offers Distributed Hash Table functionality where the different schemes are built on. The simulator collects the following metrics:

- standard deviation of estimates,
- number of messages sent,
- number of hops on the DHT for messages.

The simulator models perfect channels with no message or link loss. The simulator models churn, such that we can simulate join and leaves (dynamism) in the system.

8.1 Definition of metrics and measurement

Throughout all evaluations we will set up each measurement with the following properties:

- type of scheme to be evaluated {asymmetric, symmetric, Jelasity},
- Define size N of DHT identifier space: $N=2^b$,
- Define size of population n: $n \leq N$,
- All nodes n in population have a load between 0...100.

For all measurements we consider the objective of finding the s*ystem average load.* The load is modeled as a value between 0...100, where 0 is unloaded and 100 is full loaded. Each node will have an initial load and they are uniformly distributed. Thus the initial **true system average** load is 50.

The symmetric and asymmetric schemes are described in the algorithm section. We will show results for the *improved* version only. Jelasity scheme is taken as a reference and as objective of comparison.

Estimates represents the system average load value. The value is always weighted in our schemes, since we proceed aggregation in steps where each step is a interval of different size and population. when we speak of an estimate, we always refer to *a node which estimates the system load.*

Standard deviation of estimates represents the distribution of the estimates from all nodes in the system. The smaller the standard deviation the better the estimates do agree on the mean value in that distribution. The mean value is the *estimated* true average value.

Simulating the protocol (scheme)

The simulator proceeds in the following discretized manner (Figure 23): Each round has L cycles where $L=\log_2 N$. A round is a complete run of the protocol. This means that after a round the aggregation algorithm has finished. Each cycle represents a level. There are *n* nodes which run the algorithm in each level. In the simulation, the algorithm is run by one node at the time (respectively algorithm 1 is run by that node).

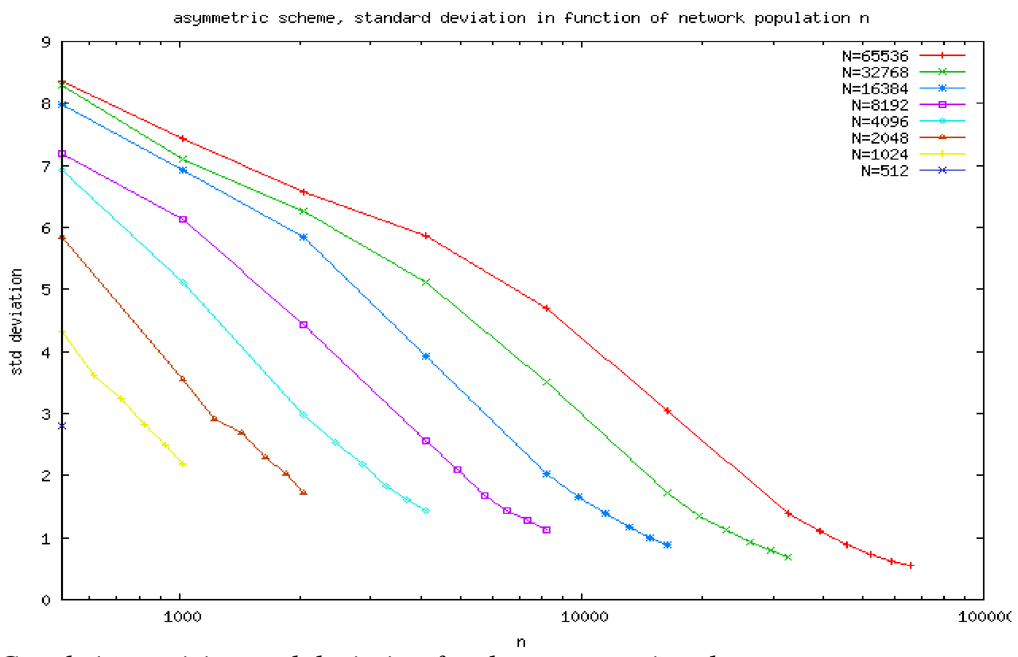
This inherently let nodes proceed synchronized, where the synchronization variable is the length of one cycle. All schemes are simulated the same way. Jelasity [45] gives a practical way of achieving synchrony. In the symmetric scheme we do not need synchrony as described earlier. In the asymmetric scheme we need synchrony, but as also described earlier we can easily overcome need for synchrony.

```
    round scope
        cycle scope
            node scope
L   n   1
            {node run algorithm 1}

            end node scope
        end cycle scope
    end round scope
```

Figure 23: Discretization of the protocol in the simulator

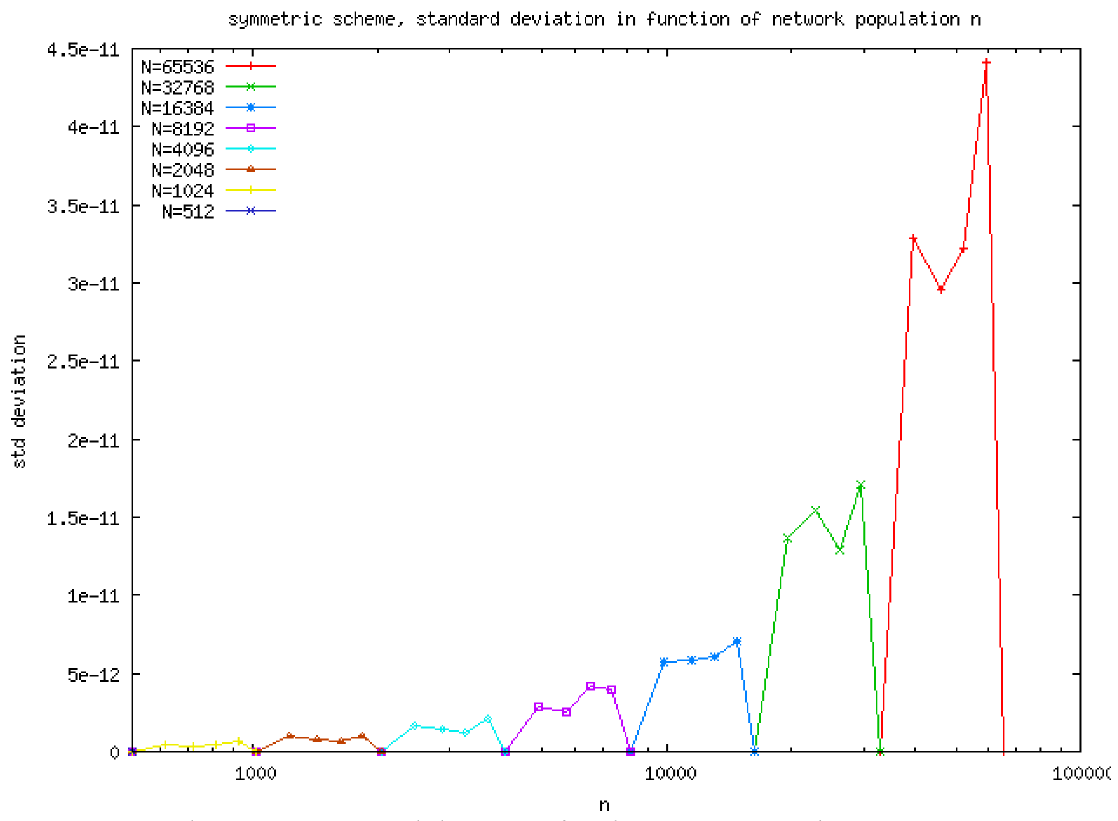
8.2 Precision of the estimates

With the precision evaluation we compare how well nodes estimate the system load. The comparison is done by measuring the standard deviation of all the estimates.

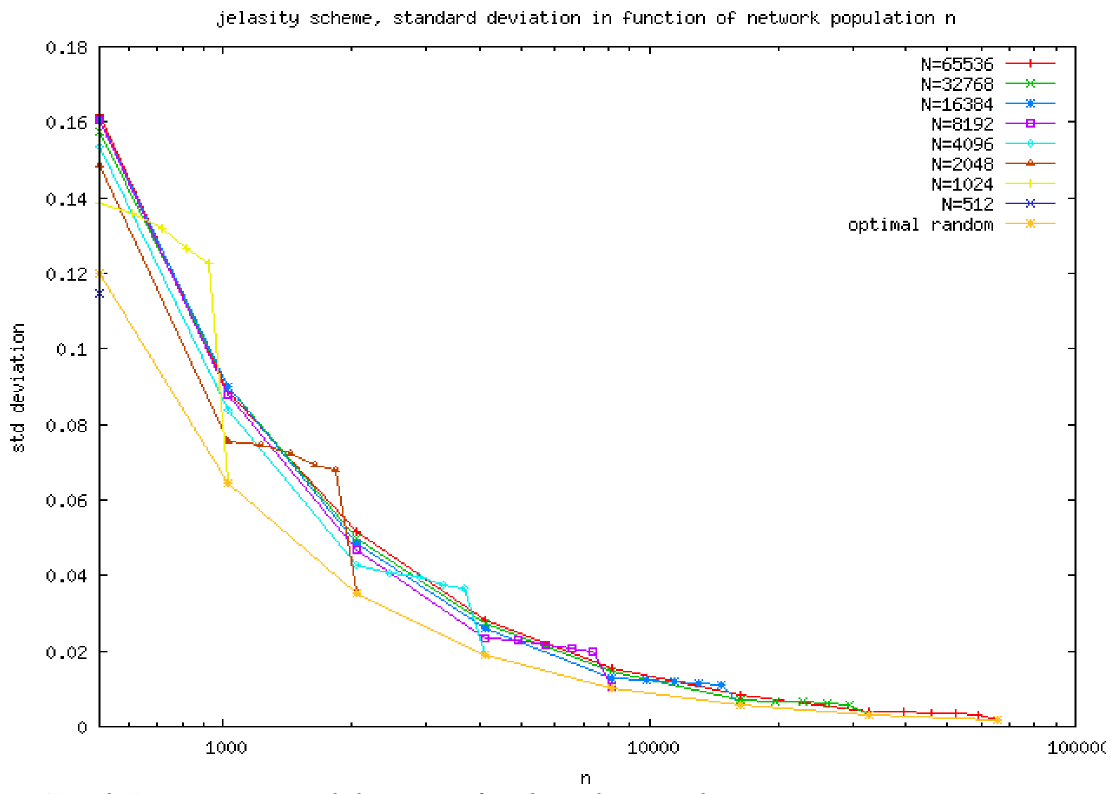


Graph 1: precision: std deviation for the asymmetric scheme

Graph 1, 2 and 3 show standard deviations of the estimates in function of the network population. Different lines represents different sizes of identifier space (N). The x-axis represents the population. The chosen populations (points in graph) are the following factors of N: 1, 0.9, 0.8, 0.7, 0.6, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/256.



Graph 2: precision: std deviation for the symmetric scheme



Graph 3: precision: std deviation for the Jelasity scheme

The asymmetric scheme, graph 1, perform worst of all three. To achieve a std deviation smaller than one is almost unfeasible for N smaller than 2^{15} in a full population.
Comparing the asymmetric scheme with Jelasity, graph 3, we never perform as good as Jelasity does.

We show in the graph 3 the curve for Jelasity of optimal randomness. Optimal random means that random choice is made from the set of existing nodes in the DHT, rather from the identifier space N. From the graph we see that for N=65536 and population > ½ N, we are very sharp to optimal random, by simply choosing nodes from the identifier space. Which means that for very large systems with good population, the randomness will not do better than optimal random. Thus, there is no special need for providing good randomness, as long the population is very large.

On the other hand, the symmetric scheme, graph 2, performs much better, than Jelasity. The reason for the good results is because for each message not received by a node it will be resent to the head of the interval.
Note that the Graph 2 has rounding errors, and therefore shows these "mountains". It can be regarded as a *0-std-deviation* for all measurement in that graph.

Note that Jelasity assumes good randomness to provide good aggregation precision. The CHORD DHT does not provide any mechanism to provide optimal randomness, since we simply choose random nodes from the identifier space N.

Anyhow, as we shown in the graphs, Jelasity provides good results even though not optimal random, with std deviation < 1/6. The symmetric scheme performs excellent with a multiple of orders better than Jelasity. Asymmetric does not provide good common estimation, unless N is big and almost fully populated. In our results, the asymmetric performs not better than a std deviation around1.

8.3 convergence

The convergence shows how fast estimation converges to a common value for all nodes. This common value is the mean of the estimates. The standard deviation to the mean shows how much the distance to mean is. Since the protocols runs in a discrete manner, we show the narrowing of standard deviation in function of cycles. Remember that after a cycle all nodes have run the algorithm 1 for a certain level. In our scheme a round consists of L cycles, where $L=\log_2 N$.

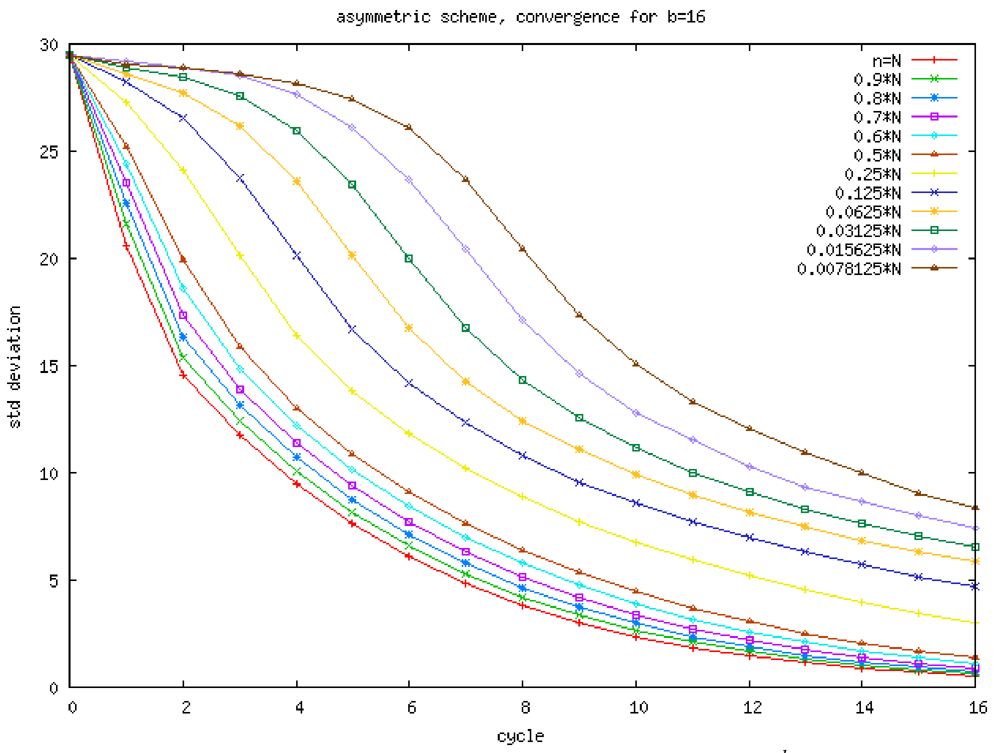
In the Jelasity scheme an epoch has jL cycles where $jL=\log_2 n$ and *n* is the population. Note that for Jelasity N is only a upper bound of nodes in the system.

We refer to epoch if we talk about the Jelasity scheme and we refer to rounds when we talk about the asymmetric/symmetric schemes.

In graph 4, 5 and 6 we show the convergence for the 3 schemes. Th DHT of identifier space is N=65536. The different lines show different populations in the DHT.

In the fully populated case the asymmetric scheme, graph 4, does not converge to a std deviation smaller than 0.55. In the case of half the population the std deviation is 1.4, and for the quarter population it is 3.

To converge to a value of std deviation smaller than 1, the fully populated DHT takes 14 cycles where for the 0.9N and 0.8N it takes 15 cycles, for 0.7N 16 cycles and the rest never goes below 1.



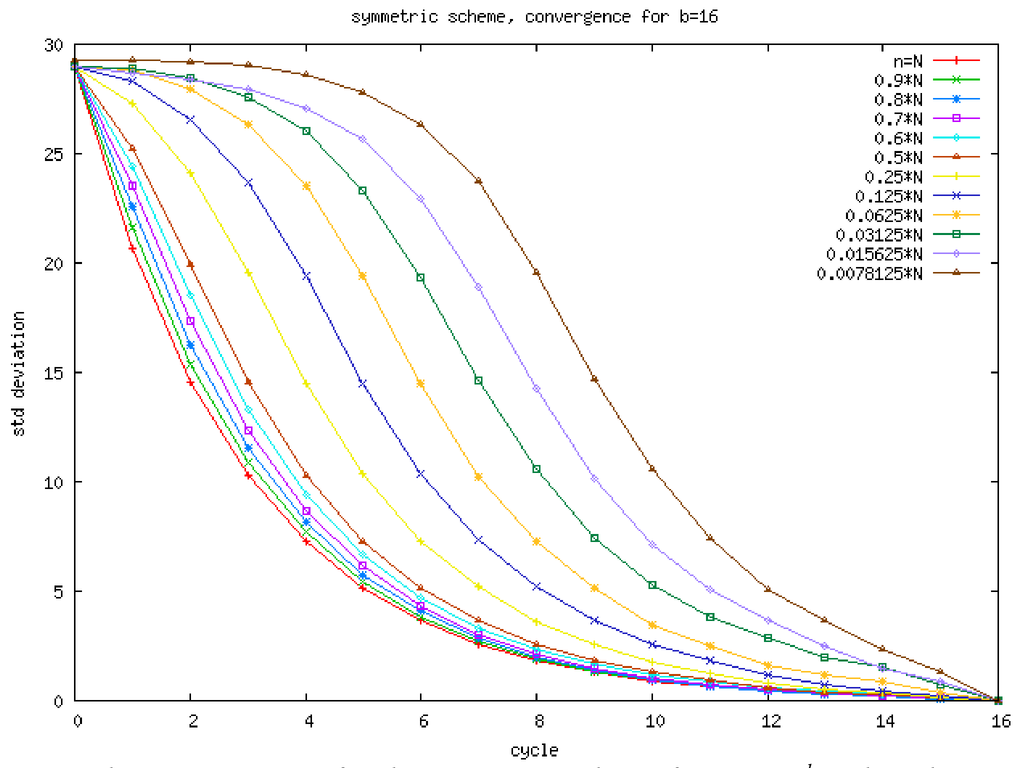
Graph 4: convergence for the asymmetric scheme $N=2^b$ *where b = 16*

The symmetric scheme, graph 5, has the nice property that in all sizes of population the std deviation will approach 0 in the L-th cycle. Compared to the symmetric scheme, all std

deviations are bound to 0 at the L-th cycle, and convergence is somewhat faster than in the asymmetric scheme.
Number of cycles needed to have std deviation <1:

population	**#cycles s.t sd<1**	**value at L-th cycle**
N	10	0.0
0.9N	10	0.0
0.8N	10	0.0
0.7N	11	0.0
0.6N	11	0.0
0.5N	11	0.0
0.25N	12	0.0
0.125N	13	0.0
0.0625	14	0.0
0.03125	15	0.0
0.015625	15	0.0
0.0078125	16	0.0

Table 2: symmetric scheme, number of cycles s.t std deviation < 1, and aggregation value



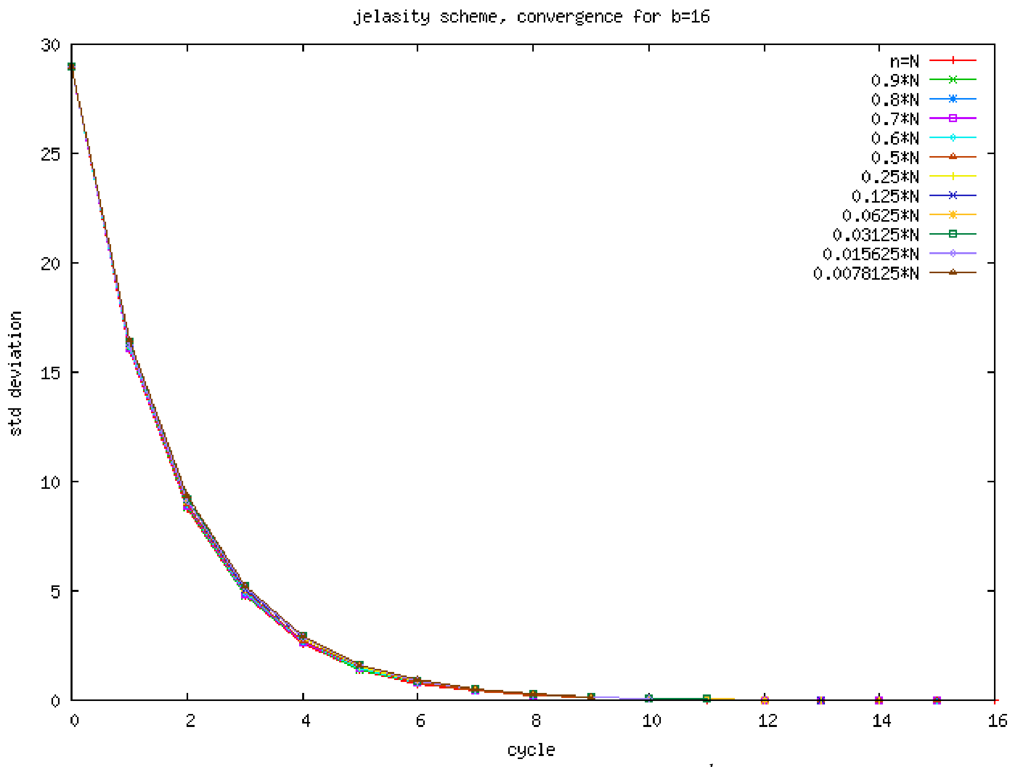
Graph 5: convergence for the symmetric scheme for $N=2^b$ *where b = 16*

The Jelasity scheme, graph 6, has a very fast convergence. As the graph shows, the convergence is independent of the size of the population. After only a few cycles the convergence is tending to 0.

population	**#cycles s.t sd<1**	**value jL-th cycle**
N	6	0.00164
0.9N	6	0.00323
0.8N	6	0.00340
0.7N	6	0.00356
0.6N	6	0.00375
0.5N	6	0.00398
0.25N	6	0.00816
0.125N	6	0.01531
0.0625	6	0.02804
0.03125	6	0.05151
0.015625	6	0.08852
0.0078125	6	0.16206

Table 3: Jelasity scheme, number of cycles s.t std deviation < 1, and aggregation value

Comparing Jelasity to the symmetric shows that for a std deviation smaller than 1, Jelasity converges 1.67 to 2.67 times faster. On the other hand, the aggregated value after the last cycle is for the symmetric scheme always 0, where as for the Jelasity scheme exists some value >0.



Graph 6: convergence for the Jelasity scheme $N=2^b$ *where b = 16. N is a upper bound.*

Jelasity has a super fast convergence. This has a significant revenue on the time to run the protocol. Jelasity terminates faster than the symmetric scheme, which makes the algorithm being more efficient than the symmetric scheme.

The asymmetric scheme does not have any good convergence properties for practical networks. A high population density mustbe present, almost N, have a high identifier space to achieve good performance. Thus making it not a good choice where all estimates need to agree close (a std deviation <1) on a value. It is up to the properties of the load balancing algorithm. If the constraint on the value of the std. deviation can be relaxed, by allowing greater std deviation, the asymmetric scheme can be applied.

8.4 DHT hops, messages and cost

In this section we discuss the results of how the message complexities of the schemes compete, and what the cost on the overlay the different schemes have.

8.4.1 messages

The graph 7 and 8 illustrates the messages sent in each scheme to accomplish a full aggregation. For the asymmetric and symmetric scheme a round consists of $L = \log_2 N$ cycles and for Jelasity a epoch consists of $jL = \log_2 n$ cycles. N is the identifier space and n is the population in the system where $n \leq N$. Hence the number of cycles of Jelasity and our schemes have the relationship $jL \leq L$.

The expected number of messages sent per cycle for each scheme is:

Jelasity	$m_j = 2\mathrm{n}$
Asymmetric	$m_a = n + \alpha \quad \alpha = \begin{Bmatrix} 0 & if\ n = N \\ (0,\mathrm{n}] & if\ n \neq N \end{Bmatrix}$
Symmetric	$m_s = 2\mathrm{n} + \alpha \quad \alpha = \begin{Bmatrix} 0 & if\ n = N \\ (0,\mathrm{n}] & if\ n \neq N \end{Bmatrix}$

Table 4: expected number of messages per cycle

Thus we have the relationship: $m_a \leq m_j \leq m_s$
n is the population. α is the number of messages which will be resent to the head of a interval, as explained in the improved algorithm. If it is not fully populated, there might be 0 resent messages or maximum n resent messages.

The total number of message, for a full aggregation, is given as the following:

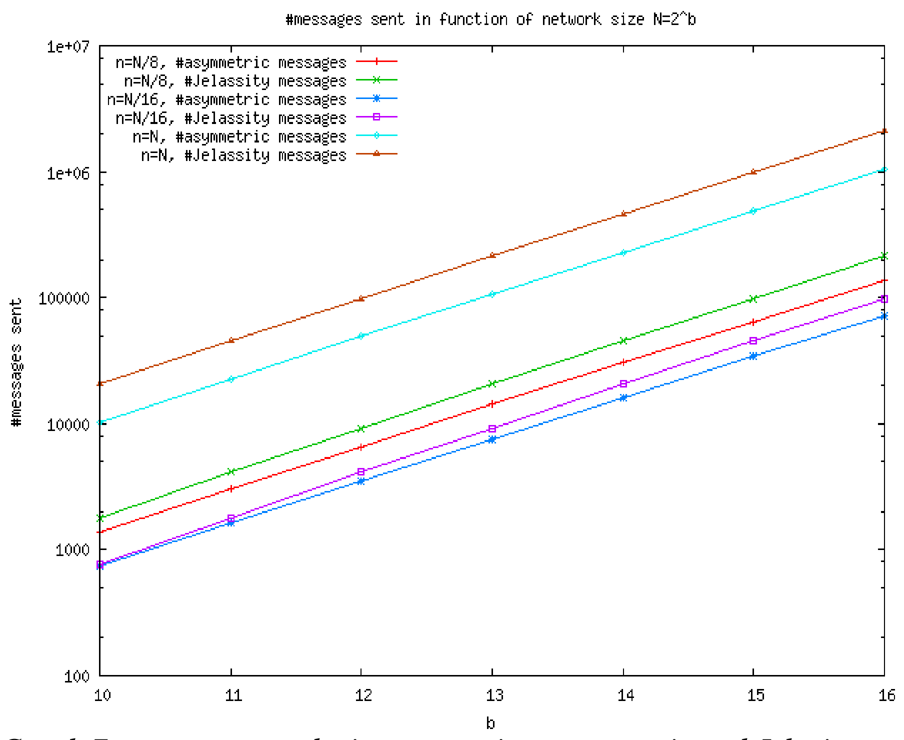
Jelasity	$M_j = m_j \cdot jL$
Asymmetric	$M_a = m_a \cdot L$
Symmetric	$M_s = m_s \cdot L$

Table 5: expected number of messages per full aggregation

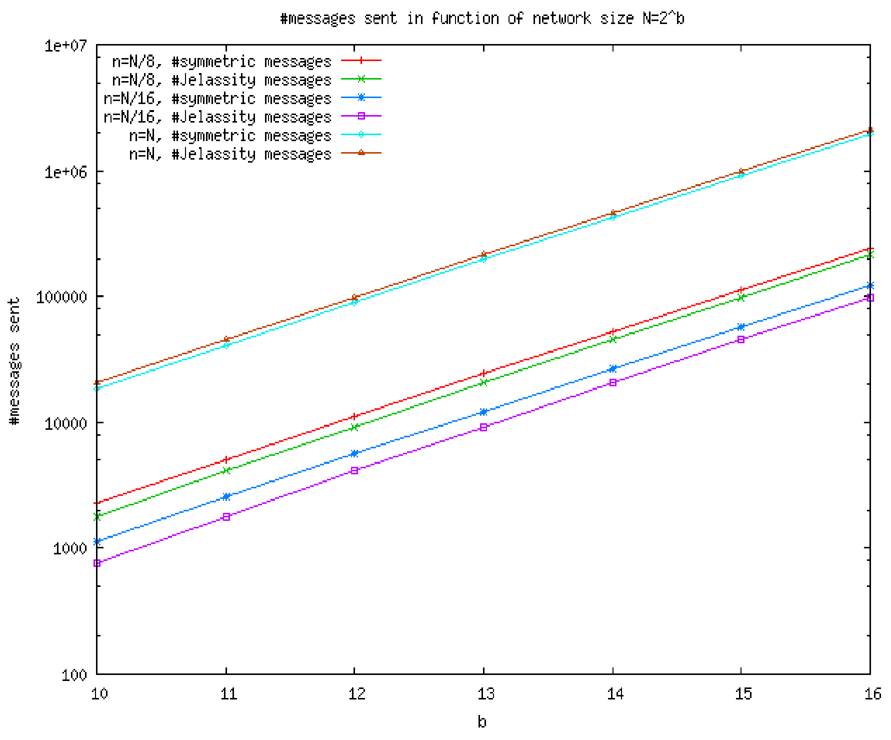
Remember that $jL \leq L$. According to our results below, we can establish the relationship between the schemes for the messages used in a full aggregation (in general):

$$M_a \leq M_j \leq M_s$$

We have implemented an algorithm which reduces redundancies on level 1 and 2 by exploiting the finger tables of the underlying overlay. There are 2 principles of reducing messages: First we do not send any messages if we know there is no node. Second we reduce messages when our neighbor already achieved the exchange. Our algorithm can be found in the code appendix. We will not further discuss it here.



Graph 7: message complexity, comparing asymmetric and Jelasity



Graph 8: message complexity, comparing symmetric and Jelasity

Graph 7 compares the asymmetric with Jelasity. The asymmetric scheme has always less messages than Jelasity. Given by our measures, table 6 gives numerical examples to the graph 7.

population	**Jelasity, #m**	**asymmetric, #m**	**factor Jelasity/asymmetric**
N=65536	2097152	1048576	2
N/8	212992	137895	1.544
N/16	98304	72501	1.356

Table 6: comparing #messages for Jelasity and asymmetric with b=16

The symmetric scheme uses twice as much messages than the asymmetric scheme. Comparing symmetric to Jelasity, graph 8, our scheme uses less messages than the Jelasity, but only if fully populated. We gain in that case from the algorithm for reducing redundancy on level 1 and 2 as discussed above.

Important to say about graph 8 is that Jelasity scheme uses less messages than the symmetric scheme. This is due to the resending of messages which have not reached any node. Even an interval is empty, nodes will resend the message according to our algorithm. This can be slightly improved: If a node receives a message targeted to a other interval, and it was originally sent to the head of that interval then a node would never resend the message since it can assume that the interval is currently empty.

population	**Jelasity, #m**	**symmetric, #m**	**factor Jelasity/symmetric**
N	2097152	1966080	1.0667
N/8	212992	243900	0.8733
N/16	98304	122160	0.8

Table 7: comparing #messages for Jelasity and symmetric with b=16

Through this results we have shown that the message complexities are $M_a \leq M_j \leq M_s$.

What can be implied is that a protocol which uses more messages than an other one will have a longer run time and it is more sensible to message loss. The statement is theoretically true, but practically we do not have any measures and discussions. Anyhow, according to the churn results, later in this section, will give us good promises that our scheme might do better than Jelasity, even messages are lost.

8.5 overlay cost

The cost of running the algorithms on the CHORD overlay are measured in number of hops it takes to deliver messages. The CHORD DHT guarantees that a message is delivered in at least $\log_2 n$ hops.
Here we give theoretically approximation of number of hops for the different schemes it takes to deliver all messages in a cycle:

Jelasity $h_j = n + n \cdot O(\log_2 n)$

Asymmetric $h_a = \frac{1}{2} n \cdot \log_2 n + \frac{1}{2} n \cdot O(\log_2 n) + \beta \cdot \log_2 n$

Symmetric $h_s = n + \frac{1}{2} n \cdot \log_2 n + \frac{1}{2} n \cdot O(\log_2 n) + \beta \cdot \log_2 n$

$$\beta = \begin{Bmatrix} 0 & if\, n = N \\ (0,\mathrm{n}] & if\, n \neq N \end{Bmatrix}$$

A node in Jelasity uses at most $\log_2 n$ hops to deliver a message to a random node and the random node will reply directly with 1 hop. Messages will always be routed in clockwise direction. This will lead us to the above formula for a gossip cycle with n nodes.

The asymmetric and symmetric scheme have the disadvantage that half of the nodes will choose a opponent in clockwise direction, which is cheap, and the other half in counterclockwise direction, which is expensive. We have made the assumption that in the latter case it will always take $\log_2 n$ steps (which in fact is an approximation).

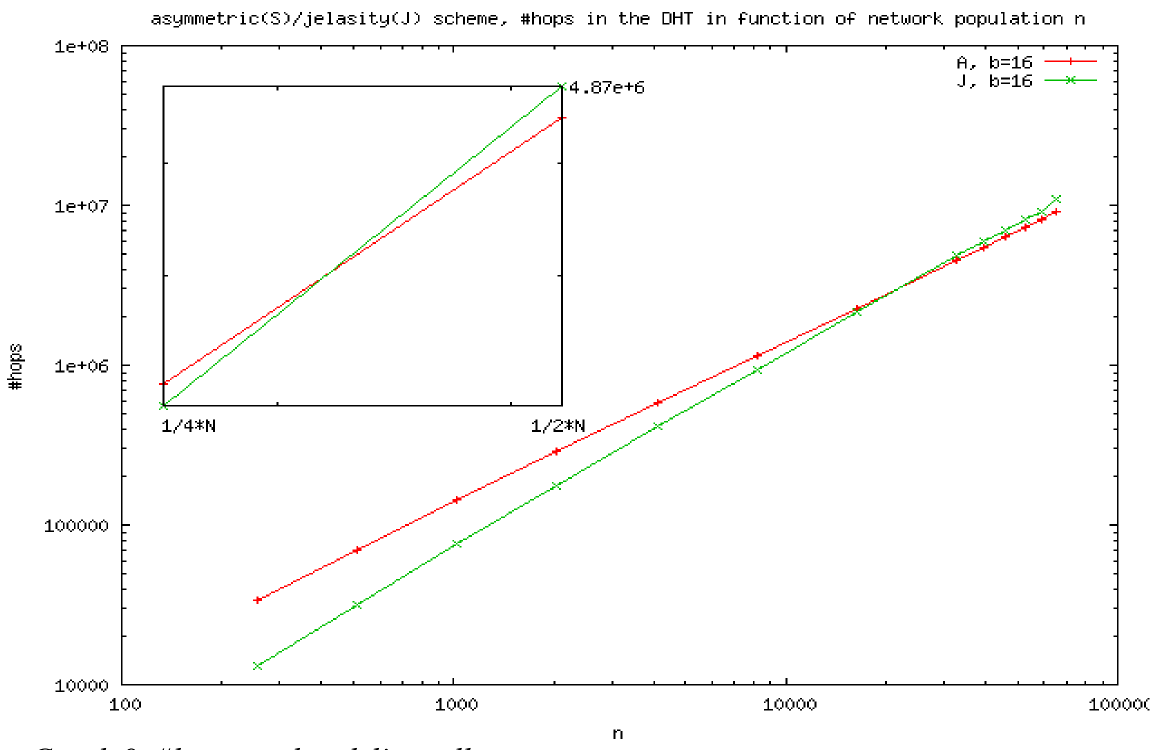
According to the improved scheme a message received by a node from an interval which was not targeted, the message is sent to the head of the targeted interval. Since it is sent in counter clockwise, we also assume for each resent message it will take $\log_2 n$ steps, in case of β.

In the symmetric scheme, a node which was sent a message to will reply directly to the originator. This is measured as in Jelasity with 1 hop.

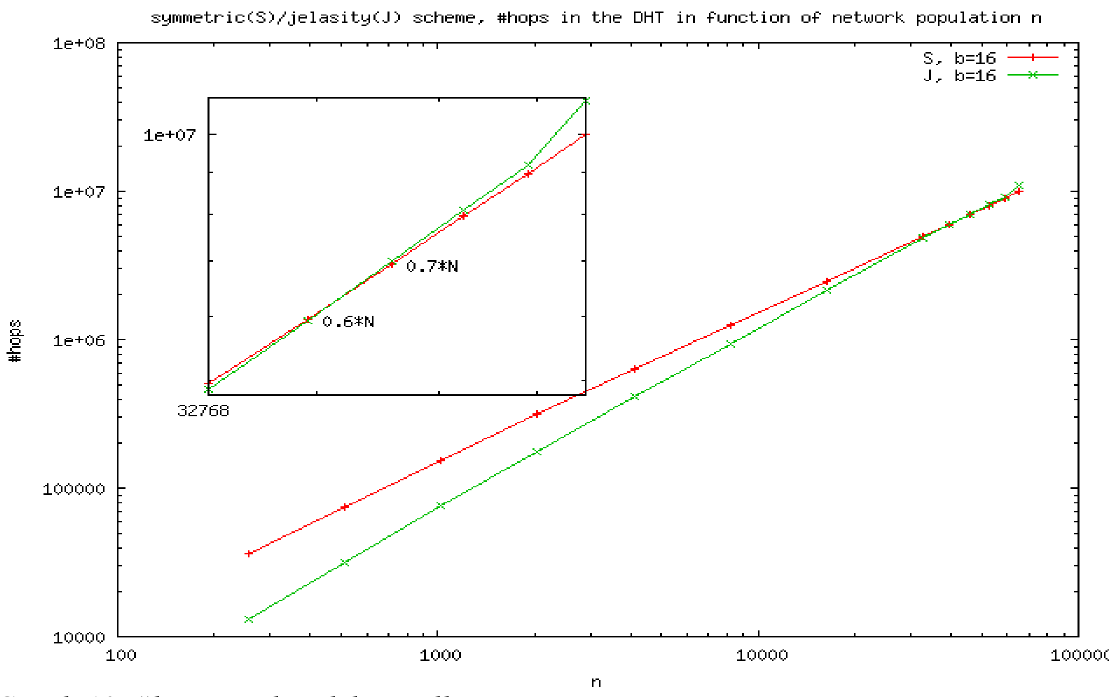
In the results, graph 9 and 10, we show how the costs on the overlay are compared to Jelasity. The asymmetric scheme and Jelasity have about the same cost in a population ~ 1/8 N. In smaller population Jelasity does much better.

population	**# hops Jelasity**	**#hops asymmetric**
N	11009573	9093510
½ N	4870400	4498393
1/32	177163	291108

Table 8: comparing #hops of Jelasity and asymmetric together with graph 9



Graph 9: #hops used to deliver all messages



Graph 10: #hops used to deliver all messages

In all measured cases, b = 10...16 can be found in appendix, Jelasity and asymmetric have always equal costs around 1/4N and 1/2N (~ 1/8N). Thus making the asymmetric scheme cheaper only for population greater than approximately 1/8, in our case for
8192< population <= 65536.

As in our theoretical assumption of the cost for the symmetric scheme we can see in graph 10 that the population must be very dense such that the cost is cheaper than Jelasity. The crossing is around 0.6N and 0.7N (~0.65N). For all simulations, b=10...16 can be found in appendix, the crossing is always between 0.6N and 0.7N.
Thus making the asymmetric scheme cheaper only for population greater than approximately 0.65N, in our case for 42598< population <= 65536.

population	# hops Jelasity	#hops symmetric
N=65536	11009573	10010284
½ N	4870400	4955578
1/32 N	177163	313577

Table 9: comparing #hops of Jelasity and symmetric together with graph 10

In a real network, the number of hops can not solely be taken to determine the time it takes to reach the destination [7]. Network latencies and node speed also matters. Also, a message taking one hop on the overlay might travel on the physical network over multiple hops. With a metric called stretch latency, overhead of DHTs can be expressed. Stretch is the factor between the time it takes to route a message from a to b through the DHT and the time it takes by directly sending message from a to b.

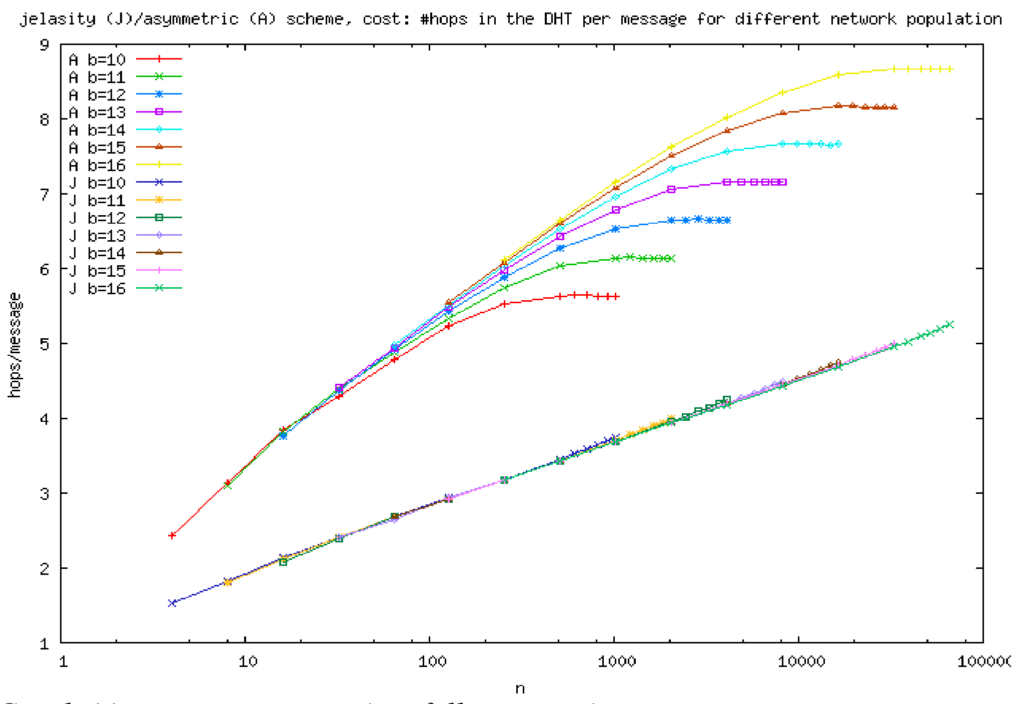
Taking stretch into account, it is crucial for having a efficient protocol by using a small number of hops.

8.5.1 overlay cost per message

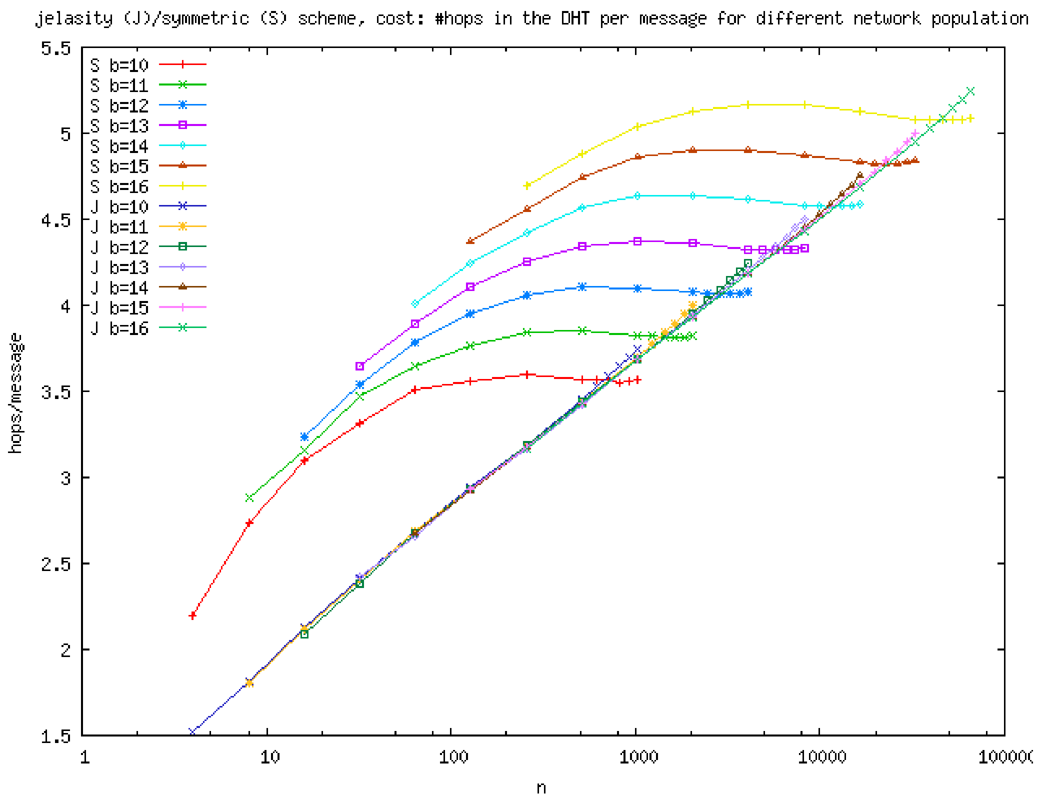
In the evaluation of messages we have established and shown the relationship of the messages sent during a full aggregation: $M_a \leq M_j \leq M_s$. The asymmetric scheme uses the fewest messages, and the symmetric the most messages. As shown just before, the cost has not the same relationship between the schemes as it is for the number of messages. We can now show the cost per message thus speaking of number of hops it takes to deliver a message. The measurements are again based on full aggregation (we use the results from the cost and number of messages).
The cost per message will tell how good the schemes does on the CHORD overlay. Of course, a cheaper cost represents a more effective usage of the overlay than a higher cost.

Graph 11 and 12 compares our scheme to Jelasity. We show for different identifier spaces the cost (hops) per message in function of the network population. The costs per message are evaluated like in the hops evaluation.

In the asymmetric scheme we have a cost per message between Jelasity and asymmetric which first increases and then saturates when population gets higher than 1/4N. Still then, the cost per message is almost double as much as for Jelasity.



Graph 11: cost per message in a full aggregation



Graph 12: cost per message in a full aggregation

In the symmetric scheme, graph 12, the cost per message between Jelasity and symmetric first increases but then gets better than Jelasity. The boarder is always between 0.6N and 0.7N for any size of identifier space (all results can be found in the appendix).

The symmetric scheme is less expensive than the asymmetric. Remember that the asymmetric scheme uses less messages, but actually at a higher cost.

By this results we see that our chosen overlay, CHORD, is not a optimal choice. The reason is because routing is done only in clockwise direction. Our schemes send half of the messages in clockwise direction and the other half in counter clockwise which is more expensive. Also each resent message in the symmetric scheme is done counterclockwise.

An improvement of the overlay can be achieved if routing can be done in both directions. Thus each node has to maintain double as much pointers. This would improve the number of hops to deliver a message, since we choose the direction which is closer to our identifier in question.
An other improvement is th choose a different overlay: For example by having a greater overlay routing table will reduce the number of hops in a lookup, because there exists a trade off between the maximum number of hops and the size of the routing table [7]. For example in DKS the system can be configured to decrease maximum number of hops by increasing routing table size. Read in 1.2.1 in [7] for details on different DHTs comparing number of hops and routing table sizes.
Investigation for a overlay achieving cheaper service to our schemes is kept as a part of the future work.

8.6 churn

Nodes might leave or join whenever they feel for it. Nodes might also fail, which in our work is considered as a leave. We evaluate the schemes under churn, which represents the schemes behavior under dynamism. Churn is defined as a disturbance by exchanging a number of nodes with new nodes. This means that n nodes are removed from the system and replaced by n new nodes. The number of nodes in the system is constant.
The measurement setup is made as the following: A churn rate defines how many nodes in the system during 1 full aggregation should be replaced. The rate is given in percentage. 0 % means no node is replaced and 100% means all nodes are replaced.

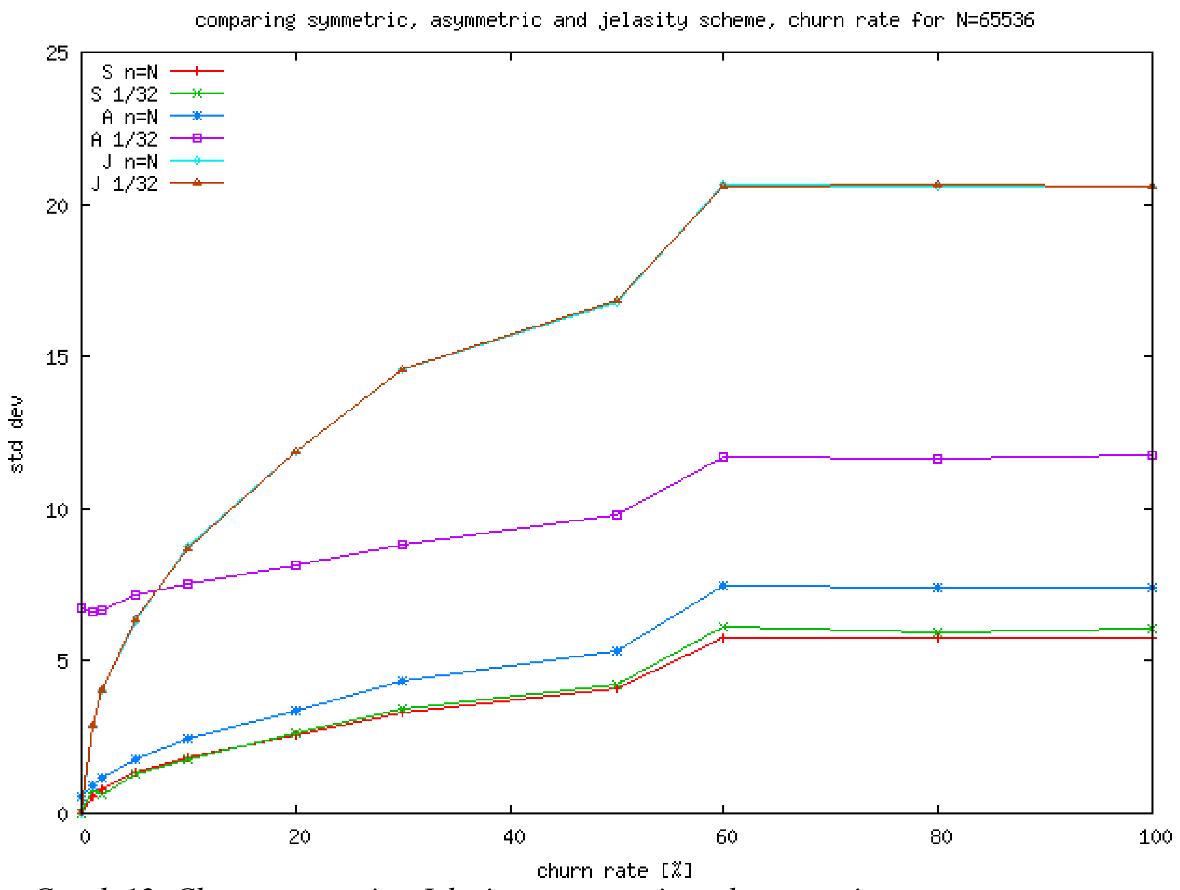
We slightly modify the discretization of the protocol in the simulator following to figure 24.

```
         round scope (R)
            cycle scope (C)
               node scope (S)

L        n     1   {node run algorithm }
                   {replace k nodes}

               end node scope
            end cycle scope
         end round scope
```

Figure 24: Discretization of the protocol in the simulator for churn

After a node has executed one step in algorithm, *k* nodes are replaced such that after $L \cdot n$ times replacing k nodes (after a full aggregation), we have replaced X% of the nodes, where X is the churn rate. Note that for Jelasity L=jL.
We implemented the replacement of nodes at the smallest discrete unit = 1 in our simulator because we guarantee in this way that our scheme is comparable to the Jelasity scheme, since this is the greatest common step in the algorithm between our scheme and the Jelasity scheme.

How new nodes participate in the protocol: According to Jelasity [45], a new node does not participate in the protocol until a new epoch (round) starts. If a new node gets contacted, it simply refuses exchange which is similar to a link loss.
Our schemes behave different to Jelasity. A new node participates from the moment on it enters the system, but it does not run the active algorithm (in the node scope) for the current level. When the next level begins it will also run the active algorithm.



Graph 13: Churn, comparing Jelasity, asymmetric and symmetric

The graph 13 compares the results for the 3 schemes. We measured the std deviation of all estimates in function of the churn rate. Per scheme are always 2 cases, one for a fully populated and one with the population of 1/32N (2048 of 65536 nodes). For Jelasity N is logically meaningless, and hence both curves overlap.

According to the graph we see that the symmetric scheme performs throughout best. Even for a churn rate of 10%, the std deviation is smaller than 2. Until a churn of 50% the std deviation increases only slowly.
The asymmetric scheme has a natural std deviation >0 if the population is smaller than N. As we see in the graph, the asymmetric std deviation curve starts at a offset of 6.71. The pro for the asymmetric scheme is that it has a flat slope increasing the std deviation till 50%.
In contrast, the Jelasity has a very steep slope and therefore a rapidly increasing std deviation even for small churn rates. At the churn rate of 1% Jelasity has a std deviation of 2.9.

churn rate [%]	**std dev symmetric**	**std dev asymmetric**	**std dev Jelasity**
0	0.0	6.71	0.00165
1	0.79	6.59	2.9
2	0.60	6.7	4.1
5	1.27	7.17	6.34
10	1.74	7.52	8.75
20	2.64	8.13	11.9
30	3.40	8.83	14.57
50	4.20	9.78	16.81
60	6.12	11.7	20.64
80	5.93	11.64	20.60
100	6.03	11.74	20.62

Table 10: std deviation in a population of n=2048 (N=65536)

It turns out that our schemes perform practically best under churn. A simple explanation is the concept of composing the aggregation value. It allows new nodes an immediate participation in the protocol even partial information is missing on new nodes (no need for synchronization as discussed in the algorithm section). New nodes in Jelasity do not participate albeit they are contacted, resulting in a link/message loss. Jelasity also needs to synchronize for each epoch such that fresh nodes can participate in the protocol.

In our schemes, removed nodes add an error to the estimates, and new nodes joining introduce an error due to lack of information. Anyhow, we showed that the std deviation is small which essentially comes from the concept of composing the aggregation.

9 Conclusion

We have studied concepts of Grid Computing and the Web Service paradigm. In the present tense of ubiquitous computing, resources for computational power can be harvested and they *are* harvested. We have explained how the Grid infrastructure managing a large-scale Internet size group of machines works, and introduced the problematic of balancing the load among the participant in a computational Grid. By analysis of fundamental P2P technologies we have introduced concepts to enable large scale and efficient self management networks.

In the survey for load balancing in Distributed Systems we analysed different solutions and compared them to each other. An introduction to request routing in the scope of load balancing lead us to the model. Job-request routing is crucial, since we believe that routing a job within a Grid should be the load balancing act.

We sketched out the model of the load balancer, based on the problem space according to the Grid4all research project. Based on the model we have introduced scalable algorithms for *n*-aggregating information in a large scale and dynamic group of nodes where nodes might fail. Through the evaluation of our algorithms, simulated in the own developed Java simulator, we have discussed the performance of the three schemes. We have compared our algorithms with the gossip-based aggregation by Marc Jelasity et. al [45], which is in our mind the most interesting to compare with.
The results of the evaluation of our schemes have shown that they are very practical and specially robust in dynamic system where node join and leave, or even die, whenever they feel for. Improvements need to be made, especially of choosing an overlay which is more effective for our schemes than the CHORD.

With this work we encourage further investigation in the usage of the aggregated values for load balancing in large scale Distributed Systems of Web (Grid) services.

9.1 Future Work

Employing and exploiting the information collected during the aggregation in load balancing is the next logical step towards load balancing. Different load balancing schemes should be exploited by using the symmetric scheme, such as comparing local load to estimate global load, or counting underutilized nodes seen during an aggregation. The algorithms can also be used to piggy back information. A vision is to enable the “power of 2 choices” [47] in a pre-emptive fashion: a request is sent to a node chosen out of 2 choices. Choices were made available during the aggregation. According to the power of 2 choices, an exponential improvement of the load in the system can be achieved when choosing between 2 at random.

The underlying overlay should be improved such that the cost of running our algorithms decreases. The overlay might also be enhanced by introducing locality awareness. If the links between nodes in the DHT are chosen regarding to their physical locality, requests might be routed in physical vicinity. Grid jobs could therefore benefit where large files must be staged in/out by reducing transfer time.

10 References

Bibliography

[1] *Globus® Toolkit 4: Programming Java Services* Sotomayor, Borja and Childers, Lisa Morgan Kaufmann 2005

[2] *A survey of peer-to-peer content distribution technologies* Stephanos Androutsellis-Theotokis and Diomidis Spinellis New York, NY, USAACM PressACM Comput. Surv. 2004

[3] *Napster among fastest-growing Net technologies* C-NET NEWS. 2000

[4] *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applicationss* Robert Morris and David Karger and Frans Kaashoek and Hari Balakrishnan ACM SIGCOMM 2001 2001

[5] *Tapestry: An Infrastructure for Fault-tolerant, Wide-area Location and Routing* B. Y. Zhao and J. D. Kubiatowicz and A. D. Joseph UC Berkeley

[6] *A Scalable Content Addressable Network* Sylvia Ratnasamy and Paul Francis and Mark Handley and Richard Karp and Scott Shenker Berkeley, CA 2000

[7] *Distributed k-ary System: Algorithms for Distributed Hash Tables* Ali Ghodsi Stockholm, SwedenKTH---Royal Institute of Technology 2006

[8] *Gnutella: Distributed Information Sharing* http://gnutella.wego.com/ 2000

[9] *KaZaA Media Desktop,P2* SHARMAN NETWORKS LTD. http://www.kazaa.com/ 2001

[10] *Explore the "Small World Phenomena" in Pure P2P Information Sharing Systems* Yi Ren, Chaofeng Sha, Weining Qian, Aoying Zhou, Beng Chin Ooi, Kian-Lee Tan IEEE Computer Society 2003

[11] *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems* ROWSTRON, A. AND DRUSCHEL, P Lecture Notes in Computer Science 2001

[12] *Peer-to-Peer Computing* Dejan S. Milojicic and Vana Kalogeraki and Rajan Lukose and Kiran Nagaraja and Jim Pruyne and Bruno Richard and Sami Rollins and Zhichen Xu HP Lab 2002

[13] *P-Grid: A Self-Organizing Access Structure for P2P Information Systems* Karl Aberer (CoopIS 2001), Lecture Notes in Computer Science 2001

[14] *Accessing Nearby Copies of Replicated Objects in a Distributed Environment* C. Greg Plaxton and Rajmohan Rajaraman and Andrea W. Richa ACM Symposium on Parallel Algorithms and Architectures 1997

[15] *Coase's Penguin, or Linux and the Nature of the Firm* Benkler, Yochai http://arxiv.org/abs/cs/0109077 2001

[16] *Web service activities* w3c http://www.w3.org/2002/ws/ 2002

[17] *IBM's web service tutorial* http://www6.software.ibm.com/developerworks/education/wsbasics/wsbasics-1-1.html

[18] *Web Services Architecture* w3c working group http://www.w3.org/TR/ws-arch/ 2004

[19] *Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001* W3C working group http://www.w3.org/TR/wsdl 2001

[20] *OASIS UDDI Specifications TC - Committee* OASIS http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec

[21] *New to Grid Computing* IBM developerWorks http://www-128.ibm.com/developerworks/grid/newto/

[22] *What is the Grid? A three point check list* Ian Foster http://www.gridtoday.com/02/0722/100136.html 2002

[23] *The Anatomy of the Grid: Enabling Scalable Virtual Organizations* Ian Foster and Carl Kesselman and Steven Tuecke Lecture Notes in Computer Science 2001

[24] *Fundamentals on Grid Computing, http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf* IBM IBM redbooks 2002

[25] *The WS-Resource Framework* Globus Alliance, IBM and in conjecture with HP http://www.globus.org/wsrf/ 2004

[26] *Open Grid Services Infrastructure (OGSI)* S. Tuecke and K. Czajkowski and I. Foster and J. Frey and S. Graham and C. Kesselman and D. Snelling and P. Vanderbilt (eds.) GGF

[27] *Globus Alliance, http://www.globus.rg*

[28] *On Death, Taxes and the Convergence of Peer-tp-Peer and Grid Computing* Ian Foster and Adriana Iamnitchi Berkeley, CA2nd international workshop on P2P Systems IPTPS'03 2003

[29] *Liste Der File Sharing Dienste* WIKIPEDIA http://de.wikipedia.org/wiki/Liste_der_Filesharing-Dienste

[30] *Load Balancing: Toward the Infinite Network* Javier Bustos-Jiménez, Denis Caromel CoreGrid TR-0049k 2006

[31] *ProActive GRID middleware* ObjectWeb Consortium INRIA

[32] *"Balancing active objects on a peer to peer infrastructure," in Proceedings of the XXV International Conference of the Chilean Computer Science* J. Bustos-Jim ´enez, D. Caromel, A. di Costanzo, M. Leyton, and J. M. Piquer IEEE Computer Society, November 2005 2005

[33] *Load Balancing in Dynamic Structured P2P Systems* Brighten Godfrey Karthik Lakshminarayanan Sonesh Surana Richard Karp Ion Stoica IEEE INFOCOM 2004 2004

[34] *Grid Load Balancing Using Intelligent Agents* Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd C&C Research Laboratories, NEC Europe Ltd., Sankt Augustin, GermanyC&C Research Laboratories, NEC Europe Ltd., Sankt Augustin, Germany, Department of Computer Science, University of Warwick, Coventry, UK

[35] *PACE --Toolset for the Performance Prediction of Parallel and Distributed Systems* G. R. Nudd and D. J. Kerbyson and E. Papaefstathiou and S. C. Perry and J. S. Harper and D. V. Wilcox The International Journal of HPC Applications 2000

[36] *Introduction into Genetic Algorithms, http://en.wikipedia.org/wiki/Genetic_algorithm* Wikipedia online Encyclopedy Wikipedia

[37] *Messor: Load-Balancing througha Swarm of Autonomous Agents* Alberto Montresor, Hein Meling, Özalp Babaoglu Departement of Computer Science 2002

[38] *CAS: Complex Adaptive Systems, http://en.wikipedia.org/wiki/Complex_adaptive_system* Wikipedia Wikipedia online Encyclopedy 2006

[39] *Anthill: A framework for the development of agent-based peer-to-peer systems* Ozalp Babaoglu, Hein Meling, and Alberto Montresor In Proceedings of the 22th International Conference on

Distributed Computing Systems (ICDCS'02), Vienna, Austria 2002

[40] *Project JXTA:An Open, Innovative Collaboration* Ra Ti On Sun Microsystem Inc. 2001

[41] *JXTA: http://en.wikipedia.org/wiki/JXTA* wikipedia.org 2005

[42] *Grid4All Grid4All (Self-* Grid: Dynamic Virtual Organizations for schools, families, and all)* http://en.wikipedia.org/wiki/Grid4all

[43] *Globus: A Metacomputing Infrastructure Toolkit* I. Foster, C. Kesselman International Jouranl of Supercomputer Applications and High Performance Computing 1997

[44] *Distributed Aggregation Schemes for Scalable Peer-toPeer and Grid Computing* Min Cai, Kai Hwang University of Southern California 2006

[45] *Gossip-Based Aggregation in Large Dynamic Networks* Marc Jelisity, Alberto Montresor, Ozalp Babaoglu ACM Tr on Computer Systemsansactions 2005

[46] *Scalable Fault-Tolerant Aggregation in Large Process Groups* Indranil Gupta, Robbert van Renesse, Kenneth P. Birman Cornell UniversityDpt. of Computer Sciende 2001

[47] *The Power of Two Choices in Randomized Load Balancing* Michael Mitzenmacher IEEE, Transactions on Distributed & Par Computing 2001