A Peer-to-Peer Approach to Enhance Middleware Connectivity

Erik Klintskog[1], Valentin Mesaros[2], Zacharias El Banna[1,3], Per Brand[1], and Seif Haridi[3]

[1] Distributed Systems Lab., Swedish Institute of Computer Science, Kista, Sweden, {erik, zeb, perbrand}@sics.se
[2] Computer Science Dpt., Univ. catholique de Louvain, Louvain-la-Neuve, Belgium, valentin@info.ucl.ac.be
[3] IMIT - Royal Institute of Technology, Kista, Sweden seif@it.kth.se

Abstract. One of the problems of middleware for shared state is that they are designed, explicitly or implicitly, for symmetric networks. However, since the Internet is not symmetric, end-to-end process connectivity cannot be guaranteed. Our solution to this is to provide the middleware with a network abstraction layer that masks the asymmetry of the network and provides the illusion of a symmetric network. We describe the communication service of our middleware, the Distribution Subsystem (DSS), which carefully separates connections to remote processes from the protocols that communicate over them. This separation is used to plug-in a peer-to-peer module to provide symmetric and persistent connectivity. The P2P module can provide both up-to-date addresses for mobile processes as well as route discovery to overcome asymmetric links.

1 Introduction

Development of distributed applications is greatly simplified by using programming systems that offer abstractions for shared state, e.g., distributed objects as in JavaRMI or CORBA. Considerable research and work has been done on protocols for shared state [1, 2], mechanisms [3], and systems [4, 5] to make them more transparent without sacrificing efficiency. The existing shared-state protocols have usually been, implicitly or explicitly, designed for connectivity-symmetric networks, e.g., LANs and clusters. By *symmetric connectivity* between two machines we understand the fact that they both can connect to each other via the physical network. On the other hand, we speak about *asymmetric connectivity* between two machines when only one of them is able to connect to the other one.

However, symmetry is not guaranteed on the Internet, in particular due to firewalls and Network Address Translators. Consequently, when the state-sharing protocols make use of messaging based on static IP addresses and assume symmetric connectivity over the Internet, they fail to work properly. In the light of this unfortunate fact, many take the view that shared-state abstractions are just not possible for asymmetric networks [6], or that new and completely different kinds of shared-state protocols are necessary. We do not share this opinion. Instead, existing shared state protocols can be directly used on top of a network abstraction layer that masks the asymmetry of the physical network.

The problem of asymmetric connectivity has been targeted at the networking layer using proxy-based architectures. Communication is routed through fixed way-points [7, 8], thus a way-point, or proxy, guarantees connectivity. This solution is static in its configuration, and requires an infrastructure for hosting the proxy. A more promising solution is to explicitly separate the name of a process from its identity [9, 10]. Name-to-address resolution can then be performed at application/middleware level, allowing for customizable strategies. This approach coincides with results from the peer-to-peer field. Organizing processes in peer-to-peer (P2P) infrastructures [11, 12], or overlay networks, has in [13] been shown to efficiently solve process mobility. However, their solution requires potentially inefficient indirection of messages and does not provide a solution for asymmetric connectivity.

The remainder of the paper is organized as follows. We continue by stating the contribution of this paper. Then, in Section 2 we introduce our middleware library. In Sections 3 and 4 we describe the design and the implementation of our abstract notion of remote processes. In Section 5 we present a P2P extension to increase connectivity for our middleware. The basic messaging performance of our middleware is evaluated in Section 6. We discuss related work in Section 7, and then conclude.

1.1 Contribution

This paper presents the design and implementation of an efficient, simple-to-use process abstraction, called a DSite. The abstraction separates the notion of a process name from its address and hides details of the underlying network by offering an end-to-end asynchronous and reliable messaging service.

The implementation of the DSite allows for simple customization of strategies for failure detection and connection establishment. This is indicated by the second contribution of this paper: the usage of P2P techniques to overcome asymmetry when establishing connections. By organizing processes in a P2P network, DSites have access to a service that provides decentralized name-to-address resolution and name-to-valid-route discovery.

2 The Distribution Subsystem

The Distribution Subsystem (DSS) is a middleware library, designed to provide distribution support for a programming systems [14]. A programming system connected to the DSS results in a distributed programming system[4]. Distribution support is on the level of language entities/data structures, over an interface of abstract entities. Associated with an abstract entity type is a consistency model, e.g., sequential consistency for shared objects. The DSS provides one or more consistency protocols for each supported abstract entity type.

[4] The system is implemented in C++ as a library and it is available for download at http://dss.sics.se

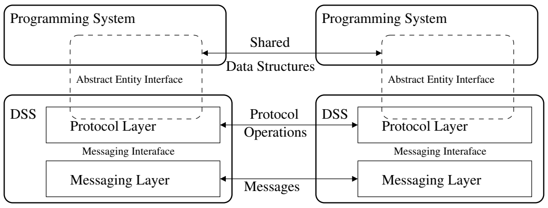

Fig. 1. The structure of the DSS middleware library. The figure depicts two processes sharing data structures using the DSS. The distribution model for the two programming systems is on the level of shared data structures. Within the DSS, the protocol layer exchanges protocol operations with other protocol instances. The bottom layer of the DSS, the messaging layer, is responsible for passing the protocol operations to the correct process.

Central in the DSS is the consistency protocol framework. This framework enables simplified development of protocols, indicated by the large suite of efficient protocols provided by the DSS [14]. The key component in this framework is an efficient and expressive inter-process service. As shown in Figure 1, the DSS is internally divided into two layers: a protocol layer that implements the consistency protocols and a messaging layer that implements all tasks related to inter process interaction, e.g., messaging. The focus of this paper is on the messaging layer. Hereinafter, we refer to a process that executes the DSS as a *DSS-node*.

At any point in time a DSS-node may know other DSS-nodes, these nodes are refereed as the *known set*. During the course of computation, references to DSS-nodes are passed among DSS-nodes, thus the *known set* changes. At any one time a DSS-node needs to communicate with a subset of the *known set*, this subset is constantly changing. Furthermore, it is perfectly possible that a DSS-node will never communicate with a given node in the *known set*. Each DSS-node is assigned a globally unique identity. In addition, a DSS-node's identity is separated from its address; this is an important requirement [10] for supporting mobile processes.

3 DSite, Representing a Process

The DSS represents known DSS-nodes as first-class data structures, called DSites. A known DSS-node is referenced from the consistency protocol module of the DSS. The task of the DSite is to provide two services: a seamless connection and communication service, and an asynchronous failure detection service. DSites can be passed within messages using the communication service of other DSites, possibly causing the introduction of a DSite in the other DSS-node. Within a DSS-node there exists at most one copy of a particular DSite.

The provided messaging service is asynchronous and guarantees reliable, in-order, message delivery (modulo failure of the receiving process). Failures are reported from the messaging layer to the protocol layer. A DSite continuously monitors the DSS-node it represents and classifies accessibility into one of the three following states:

No-problem. The DSS-node can be reached.

Communication-problem. The DSS-node is currently not accessible. This perception is local to this DSite instance. Other DSite instances, representing the same DSS-node, located at other DSS-nodes can have different perceptions. This state is not permanent, the state of the DSite can change later to No-problem or Crash-failure.

Crash-failure. The DSS-node has crashed and will never be reachable again from any DSS-node in the network. This is a global perception; all DSite instances representing the DSS-node will either be in the state Communication-problem or already in the state Crash-failure.

3.1 Channel Establishment

Within the DSS, two types of channels can be established to the node represented by a DSite. A direct channel, e.g., a TCP connection, or an internal indirect channel, called *virtual circuit*. A virtual circuit is constructed using a route of intermediary nodes (we will enter in more details in Section 5). Messages sent over a virtual circuit are passed over existing direct connections from one node to another, along the path of the route. Whether a DSite is connected directly or routed is transparent to the consistency-protocol module.

3.2 DSite API

In this section we briefly describe the interface provided to the protocol layer by a DSite and vice versa. For the reason of clarity, the interface is slightly simplified. Messages are passed as lists of appropriate data structures, e.g., integers, strings, DSites and application data structures.
send(site, msg) causes the messaging layer to transport the given message to the node identified by the given site. The protocol layer exports the following interface to the messaging layer:
receive(msg, site) called by the messaging layer when a message is received. The site argument identifies the sender of the message.
siteChangedState(site, fault) called by the messaging layer when **site** has changed its fault state[5].

4 Dividing the Labor

Realizing reliable messaging for middleware requires consideration of a multitude of requirements. These requirements include in-order messages delivery, reliable transportation, opening and closing of connections, interfacing to OS-specific services, and channel establishment. In order to efficiently fulfill them and provide a portable and extendable system, we have divided the functionality of the DSite into three separate tasks:

[5] When specialized failure detectors are used, there are provisions for turning detection on and off.

Session specific tasks. Fundamental tasks for correctness of the service a DSite provides, i.e., end-to-end message delivery. This includes ensuring reliable, in-order message delivery, (de)serialization of messages, deciding when to open connections and when to close connections.

Environment specific tasks. Connection establishment and detecting DSS-node faults tasks. These tasks are generally subject to customization, depending on the needs of the application and what the environment offers and can simply be defined as external services.

Operating system specific tasks. Link/channel tasks, that are closely related to the specifics of the operating system a DSS-node executes on, e.g., implementing socket handling.

4.1 DSite Subcomponents

The three tasks defined in the previous section are reflected in the division of the DSS into three subcomponents (see Figure 2). Session specific tasks are located within the protocol layer in the Asynchronous Protocol Machine (APM). Application specific tasks of DSite handling is located in the Communication Service Component (CSC). Operating system specifics regarding communication are located in the IO-factory. The APM is implemented as a C++ library that requires connecting to an instance of the CSC and the IO-factory. The other subcomponents are represented as abstract C++ classes in order to simplify custom implementations.

The division of the DSS into three separate subcomponents makes the middleware easier to maintain and extend. All three subcomponents interact over small, well specified, interfaces, enabling application developers to implement specialized CSC and IO-factory subcomponents independently. Furthermore, the design is in the line of separating names from addresses: the APM is responsible for the identity (the name), while the CSC is responsible for addressing, i.e., for actually establishing connections.

4.2 A Layered Approach

A DSite is realized as an extendable structure of five sub-objects, located in APM, CSC, and IO-factory (see Figure 2). Each object within the structure represents a certain task related to remote process interaction.

There are three objects located within the APM. First, the **DssSite** that provides the protocol layer interface and acts as the internal DSite reference. Second, the **Session Object** that maintains communication sessions and ensure reliable, in-order delivery even in the case of volatile connections. Third, the **Transport Object** is responsible for serializing messages[6], according to the channel type, and put serialized representations onto the channel.

Establishing connections to, and monitoring the status of, a DSS-node is assigned to the **CscSite**, located in the CSC. Establishing connections is done upon request from the **DssSite**. When a connection is established, it is passed to the **DssSite**. The actual connection has the form of a *channel* in the case of a direct connection, whereas in the

[6] in cooperation with the application, running on top of the DSS

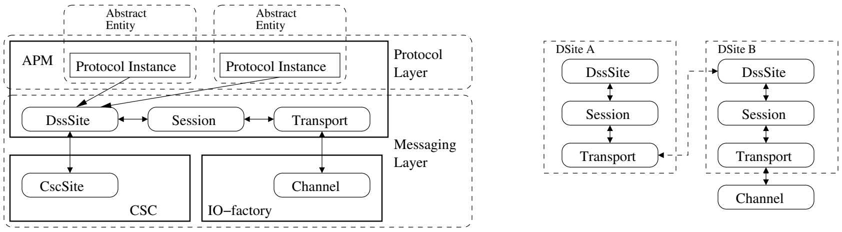

Fig. 2. To the left, the layout of the DSS and its three separate software subcomponents. The protocol layer and parts of the messaging layer are located in the same software subcomponent, the APM. The figure also depicts the location of the sub-objects that represent the DSite structure. The APM does abstract messaging and the CSC/IO-factory does the actual messaging. The figure to the right, depicts how a DSite `A` uses DSite `B` to construct a virtual circuit.

case of an indirect connection it has the form of a route, or virtual circuit, i.e., a sequence of **DssSites** describing the path to the target process. The **CscSite** is also responsible for monitoring the status of the target process; a continuous task. Detected errors are reported to the **DssSite**.

A direct connection to another process (in the form of a TCP socket, or another transport medium) is represented by a **Channel** object, located in the IO-factory. It is allocated and linked to a Transport object when a connection is established and removed when the channel is lost or closed.

4.3 Maintaining a Dynamic DSite Structure

A DSite is always represented by at least a **DssSite** connected to a **CscSite**. The other objects exist on a by-need basis, allocated when needed and deallocated when no longer needed. The Session object is allocated when the DSite is needed for actual communication, and lazily removed when there is no further communication needed[7]. A Transport object is allocated by the session object when a direct or an indirect connection to a DSS-node exists.

This design allows for a compact representation of a DSite. A DSite used only for identification is represented by a simple **DssSite** object. A disconnected DSite object with unsent messages is represented by a **DssSite** object together with a Session object.

The DSS creates DSites from descriptions commonly received along with consistency-protocol messages. DSites are automatically created when received, and automatically removed when no longer needed. Detecting obsolete DSites is done during periodic checks by a mark-and-sweep algorithm. Consequently, a DSS-node closes non-used connections automatically.

[7] Removal is partly controlled by the amount of communication needs of other DSites, i.e., resource management.

4.4 API Between Subcomponents

The interfaces that a DSite object's subcomponent interact through represent both synchronous and asynchronous functions. For simplicity basic functionality regarding identity/address serialization or connection of two objects is not described. The APM exports the following interface to the CSC, through the **DssSite**:
directConnectionEstablished(Channel), called when a direct connection has been established for the **DssSite**.
routeFound(DssSites[]) is used to inform a DSite that it should set up a virtual circuit through the sites in **DssSites[]**.
stateChange(newState), the CSC has deduced that the fault state has change and that affected protocols should be informed.

The following interface is provided to the APM by the CSC (CscSite):
establishConnection(), is called when a **DssSite** needs to communicate. Later on, the CSC will call either **directConnectionEstablished** or **routeFound**.
closeConnection(Channel), is called by the **DssSite** when no communication is needed.
disposeCssSite(), tells the CSC that the **DssSite** has been reclaimed within the APM and so should the CscSite.

A detailed description of the IO-factory interface toward the CSC and the APM is intentionally left out due to space limitations. In short, the interface can be described as a high-level socket abstraction.

5 A Peer-to-Peer Approach

P2P overlay networks implicitly offer name-based communication and routing [11, 12, 15]. The organization of the overlay network is fully decentralized. To structure the network, each participating process is required to connect to a certain number of so-called neighbor processes. As long as each participant maintains the connections it is assigned, the algorithm guarantees connectivity within the group. A P2P lookup algorithm (here we are looking for nodes rather than data) can be used to find routes between participants of the overlay network.

In this section we give a description of how the CSC is extended in our Oz-DSS[8] implementation with a P2P module to enhance connectivity, followed by a discussion.

5.1 Adding a P2P Module to CSC

The "P2P module" acts as a service for the **CscSite**, providing name-to-address resolution and name-to-valid-route discovery functionality. The module is responsible with the node management in the P2P system, and with the organization of the corresponding overlay structure for the chosen P2P protocol. The P2P module is to be used when direct connections cannot be established due to connection asymmetry in the network, or outdated addresses, e.g., in the case of mobile hosts. The resulting system provides

[8] Oz-DSS is a prototype that extends the programming language Oz with distribution support, using our DSS middleware. It is available for download at http://dss.sics.se

an illusion of a symmetric and quasi-static network over a highly asymmetric and dynamic network. The P2P module has access to all the channels opened for the APM and is allowed to open channels on its own.

5.2 A Flooding-Based P2P Module

In order to verify our approach, we implemented a simple P2P-based connection establishment schema based on flooding, similar to Gnutella (gnutella.wego.com). More efficient P2P algorithms, based on DHTs, could also be used. However, the point here is to validate the interface between the APM and the CSC.

When a **DssSite** asks its **CscSite** to establish a connection, the **CscSite** first tries to open a direct connection using the last known address. Only if that fails will the P2P module be asked for a route-and-address discovery.

The P2P module, in this flooding approach, then broadcasts a request to the neighbor set. Subsequently, they forward it to all their neighbor processes. The request forwarding ends when either the message time-to-live (TTL) expires or the target process is found. When reached, if that is the case, the target process sends its current address together with a path list along the reverse path. Upon successful return, the **CscSite** compares the received address with the one locally cached. If the address has changed (this could be the case for a mobile host), the **CscSite** tries to connect directly to the new address. If that fails, or if the address has not changed, it sets up a virtual circuit using the returned path. Thus, connectivity can be improved both in case of mobility and in case of asymmetric connectivity, e.g., hosts behind firewalls, NATs or physical network limitations in ad-hoc networks. The CSC can also try to shorten the route before handing it to the APM.

```
establishConnection(d_site)
  c_site = getCsSite(d_site)
  addr  = c_site.getAddr()
  if  ioFactory.connect(addr, channel)
    apm.directConEstablished(d_site, channel)
  else
    route = p2pMod.pathDiscovery(c_site, addr)
    if  ioFactory.connect(addr, channel)
      c_site.setAddress(addr)
      apm.directConEstablished(d_site, channel)
    else
      apm.routeFound(d_site, route)
```

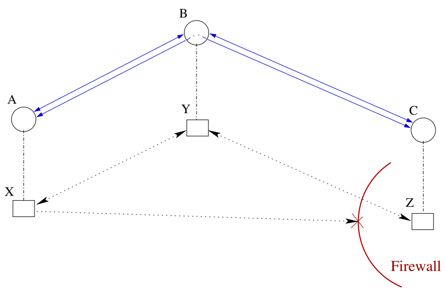

Fig. 3. Overcome asymmetric connectivity by using the name-to-valid-route discovery service. Since node `A` cannot connect directly to node `C`, it looks for a path through node `B`. On the left hand side we show pseudo-code representing the steps at node `A`.

In Figure 3 we show a simple example of three nodes, where the name-to-valid-route discovery service is used. The DSS-nodes are denoted by circles, whereas the machines they run on to are denoted by rectangles. In this example, node `A` receives, through the action of some consistency protocol, a reference to the target node `C` and tries to connect to it. As node `A` can not connect directly to node `C`, it makes use of the name-to-valid-route service provided by the P2P module, and thus it obtains a route to node `C` through node `B` which will be used to create a virtual circuit between `A` to `C` through `B`. We also show the pseudo-code representing the steps at node `A`.

5.3 Routing for Scalability

We have seen how the use of name based routing can extend the DSS functionality to cater for firewalls, NATs and mobility. In addition the P2P module can also extend the DSS with respect to scalability.

There is an appreciable cost with each direct connection in terms of memory and system resources. Thus large *known sets* are not, within reason, a problem, rather keeping a large number of direct connections might be. A DSS-node may be at the limits of its available resources and unable to accept additional incoming connections without serious performance degradation. With the P2P routing available as backup, the loaded node can now deliberately refuse additional connections requests, thus indirectly forcing the communication to take place over the already existing connections.

6 Evaluation of the DSS

The functionality provided by the DSite structure simplifies interprocess communication for the protocol layer of the DSS. However, this functionality does not come for free; it imposes a certain overhead compared to raw socket use. In this section we show that the overhead is relatively small, especially when considering Internet communication.

Table 1. The time it takes to send sequences of 1000 request-reply messages for three different applications on two different network configurations. The times normalized to the time of the socket application are shown in parentheses.

Process-Configuration	Socket	DSS – channel	DSS – virtual circuit
100Mbit LAN (ping 0.096ms)	92ms (1.0)	116ms (1.26)	287ms (2.51)
Internet (ping 51.504ms)	51073ms (1.0)	51137ms (1.0)	53404ms (1.04)

We compared a small socket application against the DSS in two settings. First, using a TCP channel for interprocess communication. Second, using a virtual circuit (over established TCP channels). All applications send a request from one process, the source, to another process, the target. Upon receiving the request, the target process sends a reply message back to the source. The reception of the reply message finishes a request-reply call. This sequence is repeated 1000 times. The tests were conducted over two network configurations: a fast LAN (0.096 ms) at SICS, and Internet (51.5 ms) setting with computers located in Sweden and Belgium. For the virtual circuit test we used one intermediary node, also in Sweden, 80km (ping 4ms) away from the source node.

The results are shown in Table 1. The socket application can be seen as the practical maximum communication speed that could be obtained (the IO-factory used by the DSS uses TCP). The overhead imposed by the DSite structure is surprisingly small, only 26%. When considering WAN settings with higher latency, the overhead is negligible. Maintenance of the virtual circuit introduces an extra 50% overhead added to the extra network hop in the LAN setting, resulting in a 250% overhead compared to raw socket communication. However, the difference is only 4% when communicating over the Internet.

7 Related Work

The JXTA specifications (www.jxta.org) define a set of basic protocols for a number of P2P services such as discovery, communication, and peer monitoring. JXTA only provides unreliable communication using the notion of pipes. Contrary to JXTA, the DSS implements reliable communication. Furthermore, the DSS is much more than a data-storage system; it is a generic middleware library supporting a wide variety of abstract entity types.

The Intentional Naming System – *INS* [16] provides resource discovery and service location for dynamic and mobile networks. The so-called resolvers in *INS* form an overlay network used to discover new services and perform late binding, i.e., a mechanism that integrates name resolution and message routing. The idea of using the lookup procedure of the P2P algorithms for routing messages in our system is very close to the late binding mechanism. However, the overlay network in *INS* is organized into a spanning tree and is intended for relatively small systems. *INS* is focused on service location, rather than process location, as is our approach. In our approach, we take advantage of the P2P lookup algorithms to extend, improve and scale up our middleware to be able to deal with asymmetric networks, mobility, firewalls, and NATs.

The research in [13] (Internet Indirection Infrastructure – *i*3) is focused on the idea of employing a P2P based overlay network to support host mobility and to provide a rendezvous-based communication abstraction. In our approach we also organize the system into a P2P overlay network. However, whereas in *i*3 the rendezvous points are used for indirection, and thus, storing extra routing state, in our proposal we use the P2P lookup algorithms for process location, directly, and message routing, indirectly. Moreover, whereas *i*3 is an independent infrastructure that has to be deployed explicitly, in our solution we propose that the very nodes of a given distributed system dynamically organize themselves to maintain communication in asymmetric networks.

8 Future Work

Currently, the DSS design is built on the assumption of a non-hostile environment. We are currently working on making the DSS a secure platform, by adding encrypted channels based on public key negotiated session keys, and unforgeable DSS-node/DSite identifiers and adresses. Furthermore, we plan to investigate and experiment more with

DHT-based algorithms in our prototype P2P component. The properties of the structured P2P algorithms, the scalability robustness, full decentralization and self-organizing make them prime candidates for our middleware system.

9 Conclusion

We described a messaging model based on a first class notion of a remote process, a DSite. The DSite abstraction hides details of the underlying network, and provides a simple to use asynchronous messaging interface. The DSS, the middleware library that implements the DSites, is designed to be both efficient and extendable. The efficiency of the design is shown in our evaluations.

The DSS matched with a suitable P2P module extends the usefulness of the DSS to asymmetric networks. The P2P algorithms work as a connection fall-back when direct connections are either impossible or resource inefficient. Direct connection establishment is not possible when dealing with mobility (mobile processes), firewalls, and NATs.

Previously our middleware (as well as other shared state systems), incorporating many state-of-the-art consistency protocols, required symmetric networks to work. In this paper we show how this limitation can be overcome by incorporating suitable P2P algorithms, greatly extending the application domain.

Acknowledgments

We wish to express thanks to Anna Neiderud and Emil Gustavsson; Anna for the work on a prototypical messaging layer and Emil for discussions about using P2P to overcome asymmetric connectivity.

This work was partially supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under IST-2001-33234 PEPITO, and by grants from the Swedish Agency for Innovation Systems (Vinnova) and the Swedish Research Council.

References

[1] Herlihy, M., Warres, M.: A tale of two directories: implementing distributed shared objects in Java. Concurrency: Practice and Experience **12** (2000) 555–572

[2] Tilevich, E., Smaragdakis, Y.: NRMI: Natural and efficient middleware. In: 23^{rd} International Conference on Distributed Computing Systems. (2003) 252–

[3] Maassen, J., Kielmann, T., Bal, H.: Efficient replicated method invocation in Java. In: ACM 2000 Java Grande Conference. (2000) 88–96

[4] Aridor, Y., Factor, M., Teperman, A.: cJVM: A single system image of a JVM on a cluster. In: International Conference on Parallel Processing. (1999) 4–11

[5] Holder, O., Ben-Shaul, I., Gazit, H.: Dynamic layout of distributed applications in FarGo. In: International Conference on Software Engineering. (1999) 163–173

[6] Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In: Mobile Object Systems – Towards the Programmable Internet. Volume 1222. (1996) 49–64

[7] Maltz, D.A., Bhagwat, P.: MSOCKS: An architecture for transport layer mobility. In: 7^{th} Conference on Computer Communications. (1998) 1037–1045
[8] Perkins, C.: IP mobility support, RFC 2002 (1996)
[9] O'Toole, J., Gifford, D.: Names should mean what, not where. In: 5^{th} ACM European Workshop on Distributed Systems. (1992)
[10] Snoeren, A., Balakrishnan, H., Kaashoek, M.: Reconsidering internet mobility. In: 8^{th} Workshop on Hot Topics in Operating Systems. (2001)
[11] Stoica, I., Morris, R., Krager, D., Kaashoek, M., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM. (2001) 149–160
[12] Ratnasamy, S., Handley, M., Karp, R., Shenker, S.: Application-level multicast using content-addressable network. In: 3^{rd} COST264 International Workshop on Networked Group Communication. Volume 2233. (2001) 14–29
[13] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., Surana, S.: Internet Indirection Infrastructure. In: ACM SIGCOMM. (2002) 73–88
[14] Klintskog, E., Banna, Z.E., Brand, P., Haridi, S.: The design and evaluation of a middleware library for distribution of language entities. In: 8^{th} Asian Computing Conference. (2003) 243–259
[15] L. Onana, S. El-Ansary, P.B., Haridi, S.: Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In: 3rd IEEE International Symposium on Cluster Computing and the Grid. (2003) 344–350
[16] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., Lilley, J.: The design and implementation of an intentional naming system. In: 17^{th} ACM Symposium on Operating System Principles. (1999) 186–201