PEPITO

IST-2001-33234

PEer-to-Peer Implementation and TheOry

Deliverable no: D4.6

Report on extended distribution subsystem

REPORT VERSION: first

REPORT PREPARATION DATE: 2004.06.30

CLASSIFICATION: Public

DELIVERABLE NO: D4.6   DUE DATE: Month 30   DELIVERY DATE: Month 30

PROJECT START DATE: 2002.01.01   PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: SICS

PARTICIPATING PARTNERS: SICS

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK

Project funded by the European Community under the 'Information Society Technologies' Programme (1998–2002)

Project Number: IST-2001-33234
Project Acronym: PEPITO
Title: PEer-to-Peer Implementation and TheOry
Deliverable No: D4.6

Report on Extended Distribution Subsystem

Due date: project month 30
Delivery date: 2004-06-30

Responsible Partner: SICS
Participating Partners: SICS

22nd June 2004

By Erik Klintskog and Per Brand

Contents

1 Introduction **3**
1.1 Motivation 3
1.2 Contribution 3
1.3 Relationship With Other Work Packages 3
1.4 Outline 4

2 A Three Dimensional Model of Home Based Protocols **4**
2.1 Coordination Strategy 5
2.2 Consistency Strategy 5
2.3 Reference Strategy 5

3 The DSS Middleware Library **6**
3.1 Abstract Entities – the API 6
3.2 The Coordination Layer 6
3.3 The Messaging Layer 7
3.4 Globally Unique Identites 7

4 Extending the DSS with a DKS service **8**
4.1 The DKS algorithm 8
4.1.1 Organizing the Overlay 8
4.1.2 Key Based Routing 9
4.1.3 Broadcasting Over the DKS Network 9
4.2 Internals of the DKS Layer 9
4.3 The DKS Instance 9
4.3.1 Naming DKS instances 11
4.4 The DSS Backbone 11
4.4.1 Persistent Storage 12
4.4.2 Joining the Backbone 12
4.4.3 The Backbone Interface 12

5 A Backbone-Powered Migrating Coordination Strategy **13**
5.1 Coordinator Location Repository 13
5.2 Migrating the Coordinator 14
5.3 Detecting Outdated Coordinator Reference 14
5.4 Distributing Coordinator Locations using Gossip 15

6 A Decentralized Stream Consistency Strategy **16**
6.1 Organization of Proxies and Coordinator 16
6.2 The Cost of A Dedicated DKS network 16

7 A KeyBased Routing Library Using the EDSS **16**

8 Conclusion **17**

1 Introduction

This report describes the extended distribution subsystem. The extended distribution subsystem(EDDS) is the distribution subsystem(DSS)[6, 8] enhanced with the implementation of a structured Peer-To-Peer algorithm [2]. In this document we show how the algorithm is integrated into the DSS architecture and made a central service component for the different modules of the unique coordination schema provided by the DSS. The algorithm is explicitly exposed as a key-based-routing [4] interface with efficient broadcasting support. Furthermore, the algorithm is used to create new strategies for two domains of shared data structure coordination. This document is not only a description of how a structured peer-to-peer algorithm can be integrated in a middleware library for distributed computing, but is also a description of how traditional middleware services can be augemented and strengthend with peer-to-peer suport.

1.1 Motivation

The DSS, with its large protocol suite[7], has been used to implement Internet applications, but is best suited for small scale applications with stable nodes. Even though the coordination framework provides the programmer with numerous distribution choices for a shared data structure, the DSS lacks protocols that scale well and prototocols that can handle unstable nodes. However, by incorporation of the latest structured peer-to-peer algorithms, the DSS can be made a middleware library for dynamic environment such as the Internet. The properties of the algorithm scalability, fault tolerance, and decentralization are, in our design, carried over to basic DSS functionality, making the consistency protocols more scalable, fault tolerant and descentralized.

1.2 Contribution

This report contributes in two areas. First, how a structured DKS algorithm can be integrated in the DSS and act as a generic service platform. Second, how a structured P2P algorithm can improve the home-based-protocol suite of the DSS.

- ▷ We describe a design and implementation of the integration of the DKS algorithm into the DSS middleware library. The result is a generic service that can be used by different activities in (and outside) the DSS. Interesting to note is the usage of the network identities to allow for multiple DKS instances to run in parallel in the same process. Moreover, the marshaling mechanism of the DSS makes it possible to construct unique textual references, aka tickets, to DKS instances.
- ▷ By design and implementation we show how a structured P2P algorithm (the DKS [10]) is used to extend our home-based protocol framework with two novel protocol instances. The self-organizing property of the DKS is used to realize a fault tolerant home-migration protocol. The scalable broadcast property of the DKS is used to realize a scalable stream protocol.

1.3 Relationship With Other Work Packages

The key-based-routing algorithm used in the DSS, the DKS algorithm, is developed in WP2 of Pepito. The EDSS is to be used by WP3 in order to create a P2P programming language, i.e. Mozart extended with P2P functionality. Furthermore, the EDSS is used in WP5 to create a scalable and decentralized Weblog application.

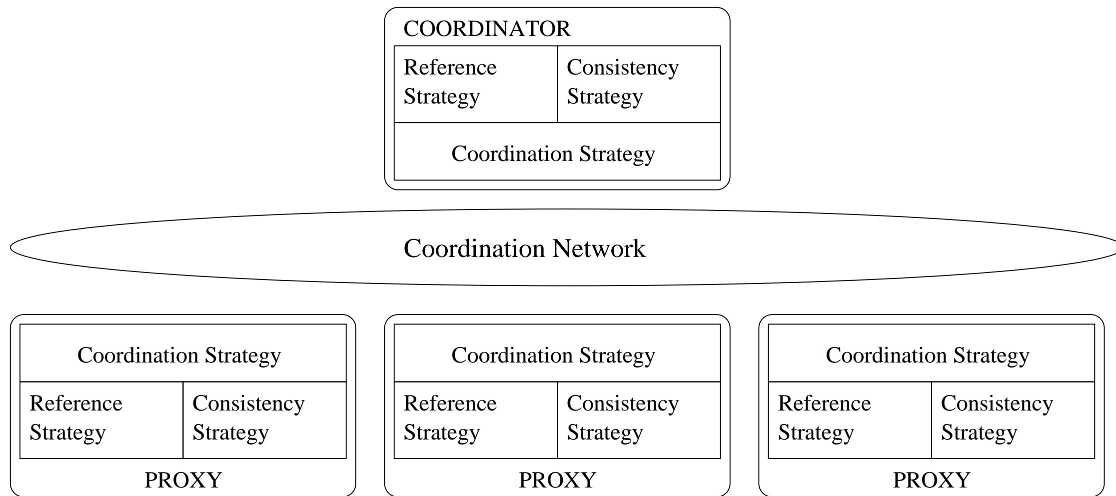

Figure 1: A coordination network consisting of three proxies and one coordinator. The coordination strategy components of the proxies and the coordinator defines how the other to components, the reference strategy and the consistency strategy communicate over the network.

1.4 Outline

The contribution of this document is the integration of the DKS algorithm into the DSS and how that algorithm is used to create new functionality in the DSS. In order to understand the new functionality in the DSS we have choosen to repeat information found in earlier deliverables (D4.9 D4.2). Section 2 and Section 3 describes the basic services provided by the DSS. Section 4 gives an overview of the DKS algorithm, and describes how it is integrated in the DSS. Sections 5 and 6 present the real contributions of the EDSS, the use of a key-base-routing algorithm in the home-based protocol framework. The external key-based-routing interface is described in section 7. The report is concluded in section 8.

2 A Three Dimensional Model of Home Based Protocols

Most protocols used for shared state in distributed systems are home-based, even though the protocols might allow for operations to be executed locally [14, 15]. A home based protocol is characterized by one and only one arbitrating unit, called the *coordinator*, and a set of reference holders, called the *proxies*. The class of home-based protocols the DSS addresses, all provides access to a shared state, under a defined consistency model, i.e sequential-, casual-, or pram-consistency [11]. Access to the shared state is in the form of abstract operations. An abstract operation can be performed either locally or remotely, it is the home-based-protocol that defines how operations are resolved. For any given home-based-protocol instance, the proxies and the coordinator form a virtual network, called the *coordination network*.

We have in WP4 of Pepito earlier reported (D4.1) the novel home-based-protocol model [8, 6] developed and implemented in the DSS. The model clearly separates functional aspects from non-functional aspects. We have successfully used the DSS to implement a wide range of different existing home-based-protocols.

A home based protocol is divided into three different functional components or strategies, see

Figure 1; the *coordination strategy* defining the communication infrastructure, i.e. how proxies and the coordinators communicate. This includes locating the coordinator and potentially moving the coordinator. The *consistency strategy* defines how the shared state is accessed, i.e. locally or remotely. It is the consistency strategy that defines the consistency model. The *reference strategy*[1] defines when the coordination network can be dismantled. Each strategy is defined as a pair of a coordinator-instance and a proxy-instance.

2.1 Coordination Strategy

The purpose of the coordination strategy is to provide an abstract communication framework for the other strategies. The framework exposes two operations, send to proxy at a process and send to coordinator. Thus, the coordination strategy relieves the other strategy-instances the burden of intra strategy communication. Furthermore, a strategy-instance is informed of perturbations to the Coordination Network, as for exampleloss of the coordinator.

The coordination strategy can express different types of coordination networks, with the same interface to the other strategies. Naturally, coordination networks with a stationary coordinator are expressed. Each coordination-strategy-proxy-instance knows the static location of the coordinator and can thus communicate with it. However, the coordination strategy can also express coordination-networks with mobile coordinators. As long as a proxy can communicate with the coordinator, this is transparent to the other services.

Whether to choose a stationary or mobile coordinator is a question of resources. The coordinator and the proxies that implements a mobile coordinator has a larger memory footprint than a stationary coordinator.

2.2 Consistency Strategy

The consistency strategy defines and implements the operations that can be performed on the home-based-protocol, e.g. send, attract state, get a read-only copy. The proxy instances and the manager instance of the consistency strategy together runs a protocol to ensure correct access of the state to the shared data structure.

The proxies explicitly control the threads that access the copies of a shared data structure. The interface allows a proxy instance to either suspend a thread, let a thread execute an operation on the local instance, or resume by letting it perform the operation on the local instance, or resume a thread and pass a result to the thread.

The proxies and the manager of the consistency strategy use the interfaces provided by the coordination strategy for their communication. Thus the choice of coordination strategy does not affect the consistency strategy. For example, a proxy unit has no notion of the exact location of its manager unit, but uses the functionality provided by the coordination network.

2.3 Reference Strategy

Manually maintaining a coherent and correct memory configuration in a distributed system is known to be hard. Instead, the problem of detecting unused data is realized by automatic memory management techniques, i.e. distributed garbage collection [13]. Different reference consistency techniques are commonly used in shared-data systems, at least when considering open distributed systems. Examples of reference consistency techniques are reference listing, and reference counting.

[1] The Reference Strategy was originally called the Memory Management Strategy.

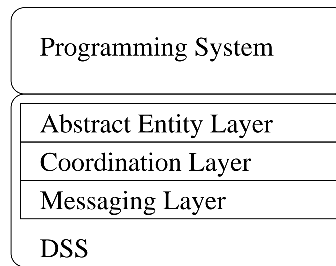

Figure 2: The DSS attached to a programming system. Note the three internal layers of the DSS.

The reference strategy is responsible for detecting when the coordination network is no longer needed, i.e. can be garbage collected. A proxy instance represents a valid instance to the shared data structure. The coordinator instance detects when no proxies exist and it is therefore safe to dismantle the coordination network.

A large number of different garbage collection algorithms exists that can be made (and to some extent are) available as Reference Strategies. Different algorithms have different properties depending on usage pattern and the environment the references will be shared in. Furthermore, algorithms can be combined to create new algorithms with the joint properties of the sub-algorithms.

3 The DSS Middleware Library

The DSS is a three layered structure(see Figure 2) intended to provide distribution support to a programming system. The topmost layer, the *abstract entity layer* provides the generic data structure interface and efficiently hides the internals of the library. The second layer, the *coordination layer*, is responsible for coordinating different copies of shared data structures according to a given consistency model (the home-based-protocols are implemented by this layer). The bottom layer, the *messaging layer*, provides communication support for the protocols of the Coordination Layer. The layer is actually available as a standalone component. The DSS is implemented as a C++ library.

3.1 Abstract Entities – the API

The API of the DSS allows for sharing of programming system data structure in the form of Abstract Entities. The DSS has no knowledge of the structure of the programming system data structures. Instead, the DSS and the programming system communicate over a set of operations, e.g.install/extract state and pass operation/result. Abstract Entities are of different types (there are three types: **immutable**, **mutable**, and **transient**), the type defines the kind of operations that can be performed on the shared data. The mutable abstract entity for example, allows both read and write access. The immutable allows for only read access, i.e. no updates. It is essential to choose the right abstract entity type for a given data structure in order to achieve efficient distribution [7].

3.2 The Coordination Layer

The coordination layer implements the home-based-protocols used by the abstract entities to guarantee consistent access to the shared entities. The different strategies are represented as objects, thus a proxy and a coordinator are both represented by three object instances; one for each strategy. This

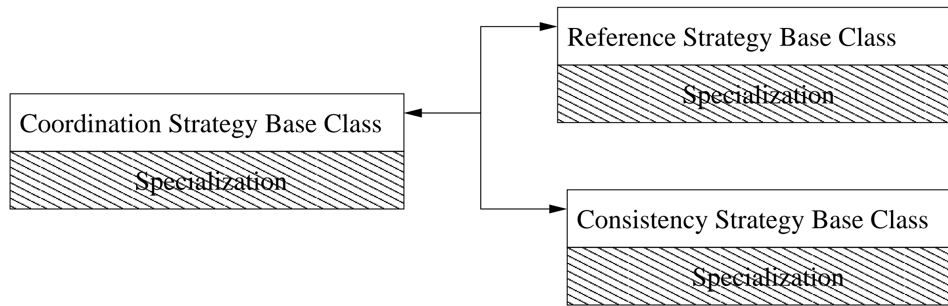

Figure 3: The internal structure of the three dimensional home-based-protocol. Each strategy is defined as a base class with some abstract methods for the specializations to implement.

is depicted in Figure 3. The DSS defines interfaces for the three strategies that defines the interfaces that each strategy implementation has to realize. Thus, a strategy instance is a specialization of a base class that by abstract methods defines the possible interaction with the other strategies.

A coordination network is uniquely identified by its globally unique name. Consequently each Proxy and Coordinator has a unique name. At a process there can at most exist one Proxy instance per coordination network (this is ensured by their unique names). New proxies are created from serialized representations. A serialized representation is either explicitly imported from file (or similar medium) or implicitly imported as a result of a distributed operation.

3.3 The Messaging Layer

The purpose of the Messaging Layer is to provide a seamless communication abstraction for the coordination layer. In order to simplify the protocols of the coordination layer all issues regarding connection management is hidden, e.g. connection establishment, connection loss, reconnection or resending of lost messages. The messaging layer exposes process identifiers, in the form of **DSite** objects. A DSite object acts as a proxy for the process it represents and provides a virtual, bi-directional channel.

Each Messaging Layer instance has a unique identity. Each DSite object contains the identity of the process messaging layer at the process it represents. When connecting a messaging layer instance, the identity is required for acceptance of the connection. This avoid malicious connection attempts and simultaneously ensures that the correct machine was found. This is essential if processes change their address binding during their lifetime. The messaging layer is described in [9] and in the Pepito deliverable D4.4.

3.4 Globally Unique Identites

Unique identities are extensively used within the DSS to identify strucures and services. Not only are coordination networks assigned unique names, but also programming system threads and data structures can be named.

A unique identity must be globally unique. Furthermore, for an identifier to be useful, it must be possible to associate values to identifiers. It must at a process be possible to check if an identifier exists, and if so extract the service it identifies. This is used when receiving a description of a service, to deduce whether an instance of the service exists, and should be used, or if no instance exists and one has to be created.

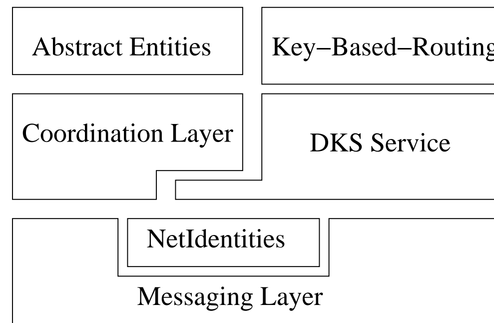

Figure 4: The internals of the EDSS, i.e. the DSS extended with a DKS service. The DKS service is located above the messaging layer and make use of the NetIdentites. Note that the DKS service is used both by the Coordination Layer and the programming system (over the key-based-routing interface).

The DSS provides an naming service, that implements globally unique names. The names are constructed from the local DSite and a locally unique sequential number. The service does also provide a generic identifier table, that stores identifiers and can be used to find local instances of a received identity.

4 Extending the DSS with a DKS service

A DKS service component is added to the DSS middleware library. The service is located above the messaging layer (as can be seen in Figure 4), it makes use of the Messaging Layer of the DSS for its communication and the identity service for identification. It provides services internally in the DSS to the strategies of the coordination networks (see Section 2). A key-based-routing interface is exposed to the programming system level. The later allows for distributed hash table implementations and alike, i.e. traditional P2P algorithm usage scenarios. This section describes the DKS algorithm, the key features used, and how the algorithm is integrated in the DSS.

4.1 The DKS algorithm

In order to clarify some design choices, this subsection gives a brief overview of the DKS algorithm. For a more throughout description of the algorithm and its properties see [5, 10].

4.1.1 Organizing the Overlay

Each node participating in a DKS overlay network is assigned a unique identifier. The identifier is used, in a structured way, to connect the node to a subset of the other nodes in the overlay. This is commonly depicted by placing all the nodes on a ring, where the nodes have pointers to other nodes located clockwise “after” the node (see fig 5 where node B is located after node A on the ring). Messages are sent over the overlay by sending messages toward the target along the local pointers (sometimes called fingers). This continues until the destination is reached. Example, if node A in the figure is to send a message to node D, A sends the message to node C that in turn sends the message to node D.

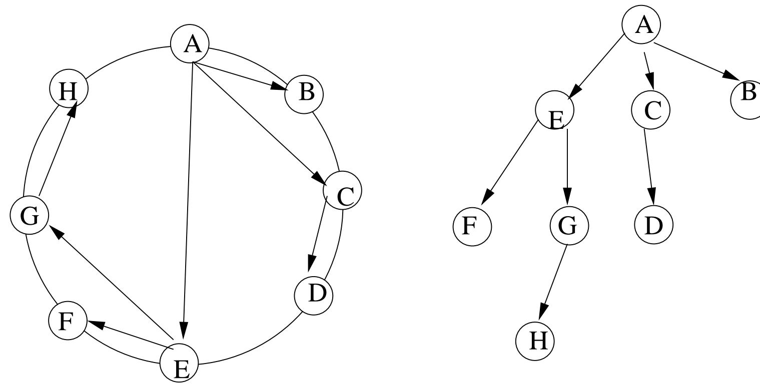

Figure 5: The overlay organization of 8 nodes. The ring depicts the organization of the nodes, together with the pointers used for node A to reach the whole network. The routing tree is depicted to the left.

Nodes can join and leave the network. To enter the network, a node needs to get in contact with a node already being member of the network. The algorithm guarantees, by correction on use, that all nodes that are members of the network can be reached, even when other nodes leaves and joins the network.

4.1.2 Key Based Routing

The DKS algorithm provides a key-based-routing service, over a defined range of keys. A message can be sent to a node being responsible for a given key using the organization of the overlay network. Each participating node is responsible for a subset of the complete range. When a node enters the overlay, it is assigned a responsible range. When a node leaves, it implicitly passes its responsibility to a node that is still member of the overlay.

4.1.3 Broadcasting Over the DKS Network

The DKS algorithm uses the implicit tree (as seen to the left of Figure 5) in order to broadcast messages to all the nodes in the overlay network. Conceptually, a broadcast message is recursively sent down the tree spanned from the initiator or the broadcast. The broadcast is guaranteed to reach all members of the overlay, without sending unnecessary messages [1].

4.2 Internals of the DKS Layer

The DKS service provides generic DKS services that can be specialized for different kinds usage. Multiple DKS instances can be run simultaneously at one process, thus one process can be a member of multiple networks at the same time. To cater for this, the DKS layer implements a repository of DKS instances, called the DKS table. The purpose of the DKS table is to elay incoming messages to the correct DKS instance.

4.3 The DKS Instance

In order to support the wide use of the key-based-routing service in the EDSS, the DKS algorithm is implemented as two components. First, the DKS Instance (see Figure 6), that runs the DKS algorithm,

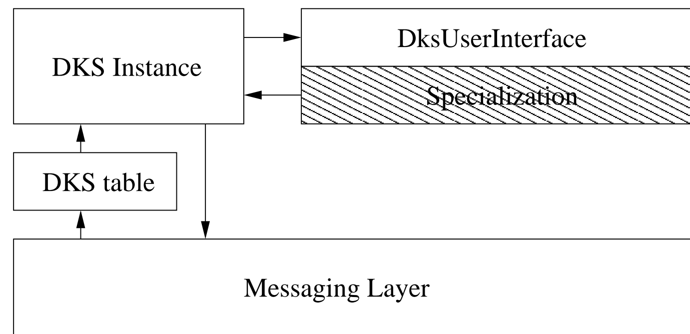

Figure 6: The Internals of the DKS service. Membership in a DKS network is maintained by the DKS Instance and DksUserInterface pair. The DKS table is repository of DKS instances that delivers incomming messages to the right DKS instance.

and communicates with other DKS instances using the Messaging Layer. Second, the `DksUserInterface`, responsible for implementing the callbacks from the `DksInstance`. The `DksUserInterface` class is abstract, it acts as an interface specification.

This design makes it possible for all different usages of the KBR service to share the same implementation of the DKS algorithm. Of course, different usages will require `DksCallback` specializations. The interface exposed by the `DksInstance` to the `DksCallback` and the interface the `DksCallback` has to implement is shown below.

```
class DksInstance{
   DKSRouteRes m_route(int key, DksMessage*);
   DKSRouteRes m_routeNext(int key, DksMessage*);
   void m_transferResponsibility(DksMessage*);
   int  m_getId();
   DKSRouteRes m_broadcastRing(DksBcMessage*);
   void m_join();
   void m_leave();
   DksStatus m_getStatus();
};

class DksCallback{
  virtual void m_receivedRoute(int Key, DksMessage*) = 0;
  virtual void m_receivedRouteNext(int Key, DksMessage*) = 0;
  virtual DksMessage* m_divideResp(int start, int stop, int n) = 0;
  virtual void m_newResponsibility(int begin, int end, int n,
                                   DksMessage*) = 0;
  virtual void dks_statusChange(DksStatus) = 0;
  virtual void m_receivedBroadcast(DksBcMessage*) = 0;
};
```

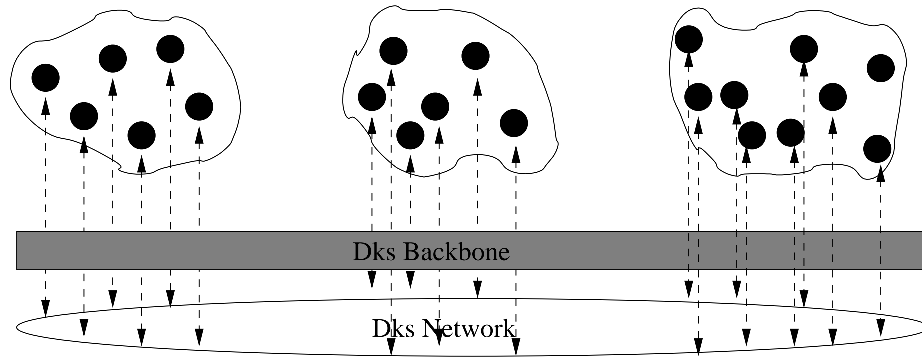

Figure 7: Organization of the DSS backbone. One DKS overlay network spans three distributed applications. An application is depicted by a cloud of black circles. Intra application communication is realized by the shared data structures. In addition, he processes(the black circles) are all members of the backbone network, not in order to communicate with each other, but in order to get access to the backbone service in the form of a persistent service.

4.3.1 Naming DKS instances

Every DKS-network is assigned a unique name when created. This name is used when sending messages between DKS instances. A message is addressed to an instance of a DKS network located at a particular process. The identity of the network is used to find the correct instance upon receiving a DKS message. The DKS table maintains a hash table that maps names to DKS instances, shown in Figure 6).

To join a DKS-network, a DKS reference is required. A DKS reference valid for joining a DKS-network can only be created by a DKS instance being a member of the DKS network. The DKS-service can from a valid DKS reference construct a DKS instance. The DKS table guarantees that there will at most exist one DKS instance per DKS network at its process. Thus, if upon reception of a DKS reference, a DKS instance for the DKS network already exists at the process, that instance is returned. Consequently, receiving the same DKS reference multiple times will only create one DKS instance.

A DKS-reference consists of a DKS-network name, parameters for the DKS-network, and a set of nodes that are members of the DKS ring. The set of nodes are used when joining the network. Note that in order to join a DKS network, the unique identity of the network is required. Because of the unforgeable properties of the NetIdentities (see D4.4), DKS references are unforgeable. Naming the DKS-networks thus automatically results in a capability-like system.

4.4 The DSS Backbone

The EDSS provides a special DKS instance that builds a backbone network. The backbone is ideally not confined to one distributed application, but spans multiple distributed applications see Figure 7. The purpose of the backbone is to provide its participants with a self-organizing, persistent storage service. Note that the backbone is a basic service for the DSS and can be regarded as a virtual network interface.

4.4.1 Persistent Storage

The DKS algorithm gives strong guarantees on the availability of key/value pairs. This fact is used by the DSS backbone. Values, in the form of *backbone-services* are inserted in the backbone network under NetIdentities as keys. In reality, there is a node in the DKS netwrk that is responsible for an key interval the key the identity of a backbone-service confines to. The backbone-service is, by the DKS algorithm, moved to that process. As nodes joins and leaves the backbone (and thus the DKS algorithm), a backbone-service will potentially move between the members of the backbone. Moving the backbone-service is a direct consequence of nodes changing their key-range responsibility. The key-based-routing functionality of the DKS allows communication with the services using their key identity, i.e. the NetIdentity.

4.4.2 Joining the Backbone

In order to get access to the DSS-backbone a node must join the DKS network that implements the DSS backbone. A reference to the DKS network is required to join the network (see Section 4.3.1). The DSS exposes interfaces for creating a backbone, creating a backbone ticket and to join a backbone, using a ticket.

When a process joins a DSS backbone it will receive a set of backbone-services to host. Concequently, the process will start receiving messages to the services it hosts. However, more importantly, the process can now make use of the backbone and insert backbone-services into the backbone.

4.4.3 The Backbone Interface

A backbone-service is a generic construction used to implement different services with high demands on persistency and availability. A backbone-service is created and inserted into the DSS backbone. The DSS backbone will transport the backbone service to the process that will host the backbone service. All interaction with a backbone service that is inserted into the backbone is done using messaging.

The DSS offers a simple interface to the DSS with two methods. One method to insert new services into the network and one to send messages to a service. `LargeMessage` is a message class that can hold a list of data items, such as DSites, integers, and LargeMessages.

```
class DssBackbone{
    void m_sendToService(NetIdentity, LargeMessage*);
    void m_insertService(NetIdentity, BackboneService*);
};
```

Backbone-services are represented as abstract classes. A particular service implementation is realized through inheritance from the service base class.

```
 class BackboneService{
  public:
    virtual void m_messageReceived(LargeMessage*,
                                   DSS_Environment* env) = 0;
    virtual LargeMessage* m_transferService() = 0;
  };
```

5 A Backbone-Powered Migrating Coordination Strategy

Home-based protocols have an intrinsic weakness in the use of a s single coordinator unit. Since machines have a tendency to either crash or be taken down the coordination network is subject to coordinator loss. In most cases, loss of the coordinator is fatal, while loss of a proxy does not affect the functionality of the coordination network. A second weakness, not as severe as the previous, is the bottleneck of the coordinator. Most home-based protocols requires communication with the coordinator to perform operations. A suboptimally placed coordinator can cause unnecessary latencies, this property is known as the tromboning effect[12]. Thus, moving the coordinator is a must for stable and efficient distributed applications.

The separation of home-based-protocols into the three different strategies(see Section 2), enables easy definition of the coordination strategy. The DSS currently has two different consistency strategies implemented, one stationary and one migratory. These two can be transparently combined with all instances of the other strategies. The migrating coordination strategy use a forward-chaining mechanism. When a coordinator moves, it leaves a forward pointer at the process it leaves. The proxies, that have a local notion of the coordinator location, uses the forwarding pointers to find the correct location. Forward chaining partly solves the problem with the single coordinator. It can be used to relocate the coordinator to a more optimal location, but the chain of forward pointers is subject to process termination and will thus potentially be broken; resulting in dangling proxies.

Three different models for locating migrating entities have been defined in the Distributed Shared Memory domain [3]: forward chaining already supported by the DSS; a persistent storage that always knows the current location of the entity; broadcasting to learn the location of a migrating entity. Realizing the third model in an open distributed environment is not only expensive in communication, but most likely impossible to realize. The second model would be possible, given persistent storage. However, if no persistent storage is available the solution does not work. Since the intention behind the DSS is to be easy to use and self contained, it is not feasible to rely on such an expensive component as a persistent storage facility for the system to function properly.

The backbone implements a virtual stable storage through self organization. Thus the EDSS can realize the third model of migrating entity discovery, using a stable storage created by the particpating nodes, in the form of a structured P2P algorithm. By storing the current physical location of the coordinator in the backbone, proxies that point to an old location can ask the backbone for the current location. In order to keep the repository in a consistent state, a migrating coordinator must update the repository to that it point the current physical location. We call the coordination strategy the Backbone Powered Migrating(BPM) coordination strategy

5.1 Coordinator Location Repository

The BPM coordination strategy implements, apart from the coordinator and proxy instances, also a backbone-service, called the Coordinator Location Repository (CLR). At creation, the coordinator instantiates an CLR instance, and inserts it into the backbone, with the identity of the coordination network as key. The CLR is initiated with the DSite object representing the process the coordinator was created at, and an epoch number, set to zero. The epoch number is increased each time the coordinator migrates and is used to order different different locations of the coordinator. Given to different location descriptions, the epoch is used to deduce which is the most recent. A coordinator location is a pair of a process and an epoch, simply denoted *location*.

At insertion into the backbone, the CLR will be transfered to the process that is responsible for the range the identity of the coordination network belongs to. From this point, the CLR can only be

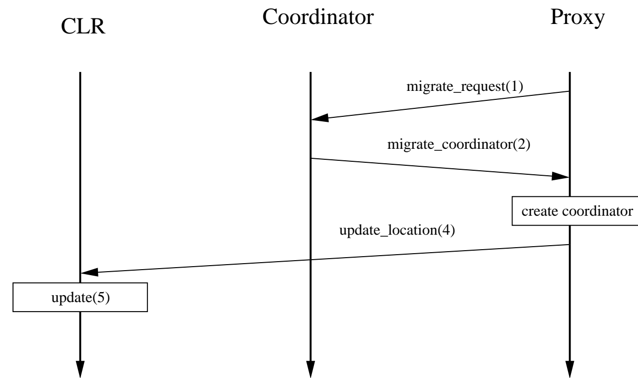

Figure 8: A sequence diagram depicting the interaction between a Proxy, the Coordinator and the CLR when migrating the Coordinator to the process of the Proxy. Note that the box labeled "create coordinator" indicates that the Coordinator has moved to the process of the proxy.

communicated with through the backbone.

5.2 Migrating the Coordinator

The proxy-instances of the BPM strategy holds the currently known location for the coordinator. The coordinator in the BPM protocol can by migration change its location. This will make the local location information at all the proxies in the coordination network obsolete. Migration, which is explicitly initiated from programming system level, is restricted to processes that holds a proxy of the same coordination network as the coordinator. In order for the proxies to be able to find the coordinator of a coordination network, the CLR has to be updated.

The interaction between the proxy that requests migration, the coordinator and the backbone unit that holds the current coordinator location is shown in Figure 8. The initiating proxy sends a migration request messages to the coordinator (1). The state of the coordinator is encoded in the migrate message sent to the requesting proxy (2). Upon reception of the migration message, a new coordinator is created, with the epoch increased by one (3). The initiating proxy is the only proxy that knows the new location of the coordinator. In order for the location discovery mechanism to work properly, a message is sent (4) over the backbone to the CLR of the coordination network (5).

Note that the coordination network has no coordinator present while the migrating message travels over the network (2-3). Furthermore, the repository holds outdated information regarding the coordinator location from (2) to (5). The coordination network will in effect be suspended until (5). However, this suspension does only effect the threads that accesses the hared data structure that is coordinated by that coordination network. Other data structures (shared or unshared) are not affected.

5.3 Detecting Outdated Coordinator Reference

The messaging layer can fail to deliver a message from a proxy to its coordinator for two reasons. Either because no coordinator for the coordination network is present at the destination process, or the destination process does not exist. Both cases are reported to the proxy-instance of the coordination strategy and in the BMP case treated as if the proxy holds an outdated coordinator location. Detection of an outaded coordinator location is lazy, only when a proxy communicates with its coordinator,

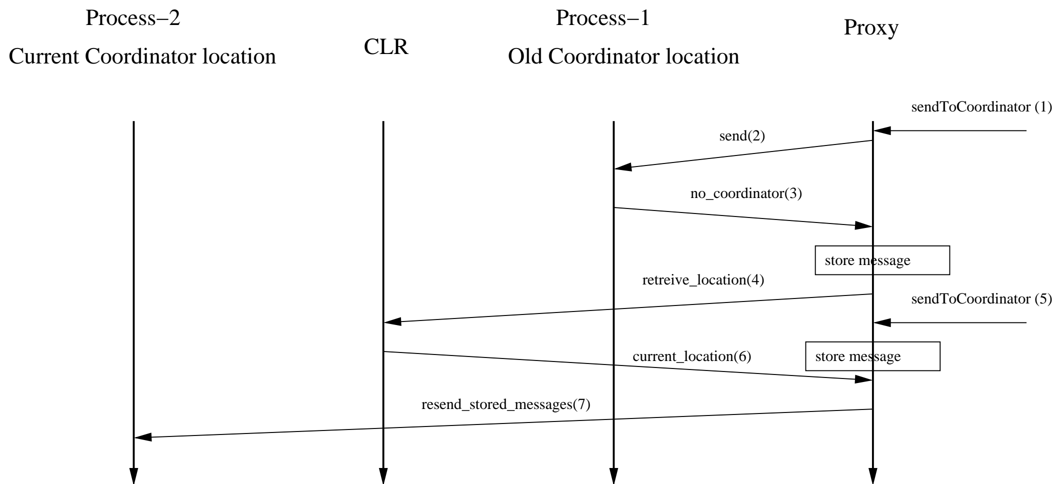

Figure 9: The sequence diagram depicts how a Proxy detects that it points to an old coordinator location. The Proxy communicates with the CLR to get the current location of the coordinator.

will it verify whether its notion of its location is correct. When discovering that the held location is outadet, the proxy will suspend further communication with the coordinator and use the backbone to find the proper location of the coordinator. No messages will be sent to the coordinator until a new location has been received. Messages sent by the other strategies to the coordinator while the proxy has an outdated location are queued, and will all be sent when a more recent location is received.

Figure 9 depicts a proxy that discovers that its coordinator has moved, and retrieves a new location from the CLR. The coordination-instance receives a message to the coordinator from the consistency strategy (1). The message is sent to the process the current location points to (2), and is returned, indicating the location to be outdated (3). The returned message is stored, and a `retreive_location` message is sent over the backbone network to the CLR (4). The consistency-instance passes the coordination-instance yet another message to be delivered to the coordinator (5). Since the proxy holds an invalid coordinator location, the message is stored. The CLR sends a message back directly to the proxy containing its current coordinator location(6). If the epoch is higher than the epoch the proxy holds, the reference is more recent and adopted by the proxy. The proxy will now send all queued coordinator messages(7) to the new process. If the epoch is older or equal to the epoch held by the proxy, the proxy sends a new request to the repository.

5.4 Distributing Coordinator Locations using Gossip

When the location of a coordinator changes, only the proxies that actively communicates with the coordinator will update their location. Since the protocols that implement the consistency and reference strategies communicate over the coordination network, that communication can be piggybacked to distribute coordinator locations. Thus, by gossiping, the load of the backbone can be reduced.

Messages within the coordination network(proxy-to-proxy, proxy-to-coordinator and coordinator-to-proxy) are attached the latest coordination location. References to Proxies sent over the network are also attached the latest known coordinator location. If upon reception of a location-epoch pair the pair describes a more recent coordinator location (has a higher epoch) the new destination and epoch is adopted, otherwise ignored. The epoch hinders a proxy from adopting locations that are older than

the location a proxy currently refers to.

6 A Decentralized Stream Consistency Strategy

The stream strategy is one of the most used consistency strategies in the DSS. It provides a many-to-many service, a proxy can send a message (or operation) that is delivered to all proxies that are members of the coordination network. The protocol make use of the coordinator as an arbitrator. The coordinator has a list of all participating proxies. Broadcasts are sent to the coordinator that in turn sends the message to all registered proxies. This solution works as long as the set of proxies is small, but does not scale when the number of proxies increases, i.e. the coordinator becomes a bottleneck.

In order to overcome the bottleneck problem, we make use of the broadcasting provided by the DKS algorithm. The DKS Broadcasting Stream (DBS) strategy constructs a dedicated DKS network that all proxies are members of. The DKS broadcast guarantees that all members of the DKS-network will eventually receive a broadcast message, nothing is said about the order of messages. However, this property is guaranteed without full knowledge of all the participants of the network.

6.1 Organization of Proxies and Coordinator

The proxies are implicitly connected by the DKS network. Broadcasts are thus not sent to the coordinator, but passed over the DKS-network. However, joining the dedicated DKS network requires a member node. This is the task of the coordinator-instance, to act as an entry point to the dedicated DKS-network.

When a proxy is created, it must first join the DKS network. A request to join the network is sent to the coordinator-part using the proxy-to-coordinator communication primitives provided by the coordination-strategy. Upon receiving the message the coordinator-part joins the proxy to the network. Later when fully joined, the proxy is informed. While waiting for ro, the proxy stores all local broadcasts. The messages are broadcast over the DKS-network when the proxy is fully joined.

Since joining is done by explicitly sending a message to the coordinator a reference to the coordination network does not contain the identity of the DKS network.

6.2 The Cost of A Dedicated DKS network

The memory footprint of a DBS proxy is large compared to a stream proxy. The stream-instance has no state at all, it make use of the coordination strategy to find the coordinator-part. In contrast, the DBS proxy holds a routing table for the DKS-network. Thus, without going into details, the DBS protocol is primary beneficial when the number of participating proxies grows so large that the stream protocol simply does not work. However, this is the strength of the DSS, the possibility to choose protocol based on the expected usage pattern. Streams that will be shared by a small number of processes should use the basic stream protocol, and streams that will be shared by large number of processes should use the DBM protocol.

7 A KeyBased Routing Library Using the EDSS

As displayed in Figure 4 and explained in Section 4 the EDSS exposes an explicit key-based-routing interface. DKS instances are represented as object instances. DKS networks can be initiated, and joined. Both operations results in the creation of a DKS instance object.

The explicit KBR interface provided by the DSS has been used in the MBlog project (Pepito – WP5) where a Weblog application is distributed over a set of processes. A hash-table layer was built on top of the EDSS, that allowed for insert, remove, update and read of data items. Furthermore, the layer provided a subscription mechanism. Processes can register an interest in changes to a particular item, if the item is altered all registered processes are informed. The publish-subscribe mechanism was realized by the use of the DBM protocol.

8 Conclusion

This report describeds how the DSS is extended with the DKS algorithm. The DKS algorithm has successfully been incorporated in to the DSS and forms a central service for the coordination layer. As a first step, a new consistency-, and a new coordination-strategy have been developed.

The new strategies have primary extended the DSS in the dimension of scalability. The combination of strategy instances caters for interesting protocols. A very interesting example is to combine all the two new strategies into one coordination network. The migrating coordinator would make the coordinator always accessible and the scalable broadcasting consistency protocol would enable a large number of participants.

References

[1] Luc Onana Alima, Ali Ghodsi, Per Brand, and Seif Haridi. Multicast in dks(n, k, f) overlay networks. In *OPODIS 2003: 7th International Conference on Principles of Distributed Systems*, 2004. To appear.

[2] Luc Onana Alima, Ali Ghodsi, and Seif Haridi. A framework for structured peer-to-peer overlay networks. Technical Report T 2004:09, Swedish Institute of Computer Science, June 2004.

[3] Jae Woong Chung, Kyu Ho Park, and Daeyeon Park. Moving home-based lazy release consistency for shared virtual memory systems. In *1999 International Conference on Parallel Processing*, September 1999.

[4] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *IPTPS '03*, February 2003.

[5] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. A framework for peer-to-peer lookup services based on k-ary search. Technical Report T2002-06, SICS Technical Report, 2002.

[6] E. Klintskog, Z. El Banna, and P. Brand. A generic middleware for intra-language transparent distribution. Technical Report T2003:01, Swedish Institute of Computer Science, January 2003.

[7] E. Klintskog, Z. El Banna, P. Brand, and S. Haridi. The design and evaluation of a middleware library for distribution of language entities. In 8th *Asian Computing Conference*, Dec. 2003. To appear.

[8] E. Klintskog, Z. El Banna, P. Brand, and S. Haridi. The dss, a middleware library for efficient and transparent distribution of language entities. In *Thirty-seventh Annual HAWAI'I INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES*, Jan. 2004. To appear.

[9] E. Klintskog, V. Mesaros, Z. El Banna, P. Brand, and S. Haridi. A peer-to-peer approach to enhance middleware connectivity. In *OPODIS 2003: 7th International Conference on Principles of Distributed Systems*, 2004. To appear.

[10] P. Brand L. Onana, S. El-Ansary and S. Haridi. 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*, pages 344–350, May 2003.

[11] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[12] M. C. Ng and W. F. Wong. Orion: An adaptive home-based software distributed shared memory system. In *Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, pages 187–194, July 2000.

[13] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), September 1995.

[14] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[15] Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. Lecture Notes in Computer Science, vol. 1686. Springer Verlag.