Internal Design of the Distribution Subsystem (DSS)

Erik Klintskog

April 28, 2005

Swedish Institute of Computer Science, Kista, Sweden

Contents

1 Introduction

This document describes the implementation of the Distribution Subsysten(DSS). The DSS provides distribution support for programming systems and is intended to be used as a middleware when creating distributed programming langue implementation. and providing distribution support on data structure level. The implementation of the generic interface is descried, together with a description of the customizable distribution strategy framework of DSS. The descriptions are on the level of C++ classes.

The DSS provides a generic service for programming systems, the middleware is designed to provide distribution support to programming systems of many differentr programming paradigms. The middleware has been succefully coupled to programming systems of the object oriented, the functional and the declarative-concurent programming paradigms. The design goal of DSS is reflected in the generic and expressive interface provided towards a programming system. The interface is expressive in that it allows for close integration with programming system constructs and it captures the behavior of the programming system. The interface is generic since it does not assume on a particular implementation of the programming system. The implementation of the interface towards a programming system is presented in the report.

The distribution service provided by DSS for programming systems is efficient and allows for a high degree of customization. Distribution support is on the level of programming system data structures, called language entities. A language entity is provided distribution support by a distribution strategy. The implementation of the distribution strategies is presented in the report.

1.1 How To Read This Document

The description of the DSS found in this report is on a detailed level and knowledge about the concepts behind the DSS are assumed. This document describes how the concepts of the DSS are implemented. It is assumed that the reader has some understanding of the DSS. This document is a complement to previously published material where conceptual descriptions of the DSS can be found. The published papers are:

- *The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities*
  The paper describes the API and the associated semantic model provided by the DSS. The internal structure of the DSS is also briefly described on a conceptual level.

- *The Design and Evaluation of a Middleware Library for Distribution of Language Entities*
  The paper presents the DSS from another perspective than *paper 1*. The focus is on the performance evaluation, it is shown that the modular design did not introduce any notable performance penalties and that the correct choice of distribution support os the key to efficient distributed applications.

- *A Peer-to-Peer Approach to Enhance Middleware Connectivity*
  The paper describes the structure of the messaging layer of the DSS. The component based design enables simple customization of connection establishment strategies. This is illustrated by the use of a simple P2P algorithm (Gnutella-like) to find suitable route between processes even when direct connections are hindered by firewalls, NATs, etc.

The DSS is implemented in C++, thus knowledge of C++ and object oriented techniques are assumed. The internals of the DSS are described on the level of the classes implementing the different concepts. For the sake of simplicity, focus is on the methods which implement the interfaces between different classes. Auxiliary and private methods are not described. Furthermore, some of the classes implement different services. In such case, parts of the class definition are introduced where the functionality is described. Below is an example of how the methods of the `Example_Class` class are introduced.

```
class Example_Class
public:
  void example_method_1();
}
```

Above is the introduction of `example_method_1` and the method is further described here: "The `example_method_1` is called ...".

```
class Example_Class
public:
  void example_method_2();

}
```

In another context of the text is a second method of the `Example_Class` class is introduced together with a description: "The `example_method_2` is". Note that both methods belong to the same class, but for the sake of convenience they have been introduced at different location in the report.

1.2   Outline

The next section, Section 2 describes the layout of the DSS. Section 3 describes the messaging layer of the DSS. Section 4 describes the abstract entity interface and the interfaces required by a programming system. Section 5 describes the coordination layer that implements the coordination protocols that realizes the shared data structure service of the DSS.

2   The Distribtion Subsystem

Internally the DSS is hierarchically divided into three layers. The topmost layer, the abstract entity layer, provides a generic shared data service. The middle layer, the coordination layer, implements the protocols necessary for the shared data service. The bottommost layer, the messaging layer, implements the communication primitives for the coordination layer. The messaging layer is provided as a standalone component that can be used outside the scope of the DSS.

Figure 1 depicts a programming system connected to a DSS instance. The figure shows a schematic picture of the internal layers of the DSS as well as the glue layer of the programming system. Furthermore, the key components for realizing transparent distribution of data structures are depicted.

2.1   The Library

The DSS is implemented in C++ and for downloaded from `http://dss.sics.se`, and compiled using gcc 3.2 under Linux. The DSS is a passive component that reacts to external events, i.e. I/O activity and operations on shared data structures. Moreover,
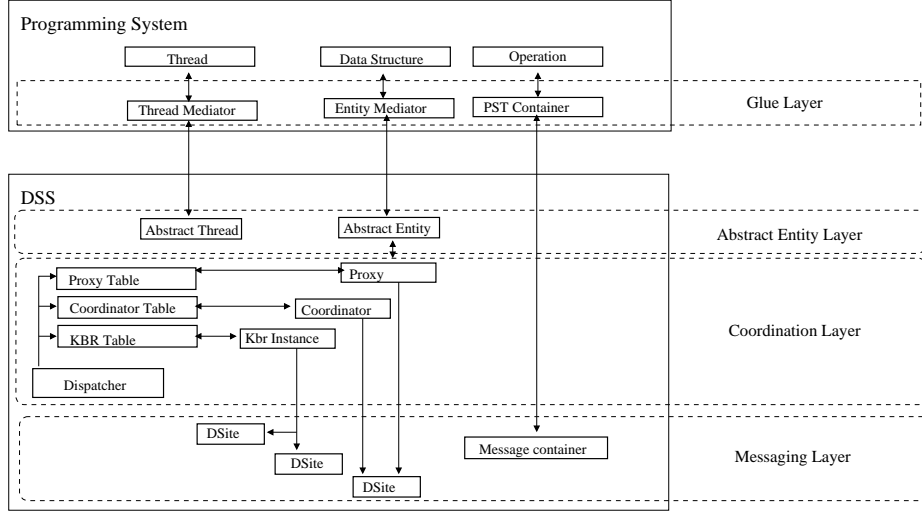
Figure 1: The figure depicts a programming system connected to the DSS. The internal structuring of the DSS and the programming system are shown, as well as the key components of both systems.

the DSS is non-blocking, a thread that invokes a DSS routine will not be directly suspended. Instead, suspension of threads is delegated to the programming system. To simplify the design of DSS, the DSS is not thread safe.

2.2 The DSS Object

In order to simplify integration of a programming system and the DSS, the DSS is represented as a class. The callback interface required by the programming system (implemented by the glue) is also represented by a class. Thus, the box denoted DSS in Figure 1 is one object, that acts as a factory for the *global threads* and *abstract entities*. Representing the DSS as an object serves as an effective encapsulation of DSS internals. Furthermore, this allows instantiating of the DSS on demand. The DSS is instantiated first when a distributed programming system actively participating in a distributed computation.

3 The Messaging Layer Library

The purpose of DSS Messaging Layer(DMSL) is to provide an efficient point to point communication service that hides the details of the underlying network. Central in DMSL is the representation of a process in the form of a first class object, called a DSite. References to DSite objects can be passed between processes. Reception of a DSite reference results in the construction of a local DSite object at the receiving process. The existence of a DSite allows for seamless communication with the process the DSite represents[1].

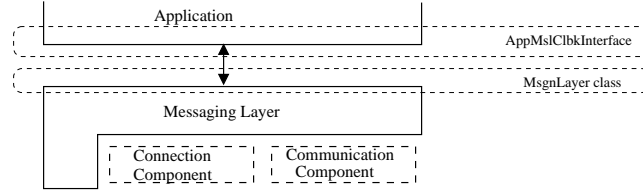[1] given that a connection *can* be established.

Figure 2: The structural layout of the DSS Messaging Layer (DMSL) library. The library provides communication service to an application. The interface between the messaging layer and the application is in the form of classes, depicted by the dashed boxes. Internally, the DMSL makes use of two replaceable components for the tasks of connection maintenance, and interprocess communication (IO).

Obviously, the underlying network can not be completely abstracted away since process termination and network perturbations can prohibit communication between two processes. Failures detected by the DMSL is categorized according to an abstract model and reported to a higher level, that is the application that make use of DMSL. In the case of the DSS middleware, the application is the coordination layer of DSS. However, as long as possible DMSL will try to deliver messages to remote processes.

The DMSL is designed to be extendable with respect to connection management and low-level I/O-handling. From the highest level of abstraction, the messaging layer makes use of two replaceable components (see Figure 2). The *I/O service* provides a low-level channel service, similar to a socket interface. This efficiently abstracts away operating system dependant properties of the communication. The *communication service* realizes connection establishment to other nodes and connection monitoring. Thus, it is the communication service that detects and classifies failures[2].

Similarly to the DSS the DMSL is represented as an object (see section 2.2) intended to be coupled to an application. The DMSL connected to an application is depicted in Figure 2. The application is supposed to implement a callback class, in order for the DMSL to communicate with the application. Examples of DMSL to application communication are received messages and detected remote process failures. The DMSL, apart from being the messaging layer of the DSS, is available as a standalone component that can be used as a messaging middleware.

This section describes the interface to the DMSL and the exposed classes. The DMSL is described as a black-box, the internals of the layer is not revealed.

3.1 The Messaging Layer Interface

The key component in the DSS Messaging Layer (DMSL) is the *DSite*, a first class process representation. The DSite is exposed outside of the DMSL and provides an abstraction of the process it represents. A DSite is primary used as a channel to the process it represents. Moreover, DSite objects can be passed by reference between different DMSL instances. All processes running DMSL instances which are known to the messaging layer are represented by a DSite object. In adition, the DMSL holds one DSite object that represents its own process.

---

[2]Lifting out functionality from the core of the system into customizable modules increases the applicability of the DMSL. Since failure detection is strongly correlated to application specifics, failure detection strategies that works for one application can make another application not work at all.

```
class DSite{
public:
    virtual void m_marshalDSite(DssWriteBuffer*) = 0;
    virtual bool m_sendMsg(MsgContainer*) = 0;
    virtual DSiteState   m_getFaultState() const = 0;
    virtual void m_mark() = 0;
};
```

Above is the class definition of the DSite object. The messaging service provided by the `m_sendMsg` is asynchronous, FIFO and reliable. Details regarding connection establishment, temporal loss of connectivity, and resend of lost messages are not exposed. Instead, problems that are impossible to hide, longer loss of connectivity and loss of the destination process are reported as a failure. However, a DSite tries to deliver messages until the target node is lost.

A process can only construct a DSite object from a proper DSite object description. Such a description for a given DSite can only be created by a DMSL process that holds an instance of the particular DSite. Process communication requires a DSite object that represents the process, thus, a DSite reference has a capability-like property. DSite objects are instantiated from serialized descriptions. Below is the definition of the messaging layer class:

```
class MsgnLayer{
private:
    _msl_internal::MsgnLayerEnv *a_mslEnv;
public: //******************DSites, marshaling and identities
    DSite*          a_myDSite;
    DSite*          m_unmarshalDSite(DssReadBuffer* buf);
    MsgContainer*   createAppSendMsgContainer();
    void m_gcResources();
}
```

There exists one DSite object instance for every known/referred DMSL process. Consequently, comparing two DSite references for equality is done by checking for pointer equivalence. The DMSL is responsible for removing DSite objects that are not used, implemented by a mark and sweep mechanism. The `m_gcResources` method schedules any non marked DSite for removal (see Section 3.6).

Figure 3 depicts the interaction between the DMSL and an application. DSML provides an interface for unserializing DSite objects that takes a serialized description of a DSite object return a pointer to a proper DSite object. A DSite object exposes an interface over which messages can be sent, the status of the process is returned, and a serialized representation can be retrieved. Communication with the application from the messaging layer is realized by the application callback interface that the application implements. Reception of messages and state (failure) of DSites is reported over this interface:

```
class AppMslClbkInterface
{
public:
    virtual void m_messageReceived(MsgContainer* const msgC,
                                   DSite* const sender) = 0;
    virtual void m_stateChange(DSite*, const DSiteState &) = 0;
    virtual void m_unsentMessages(DSite* s, MsgContainer* msgs) = 0;
    virtual ExtDataContainerInterface* m_createExtDataContainer(BYTE) = 0;
};
```

3.2 Messaging

A DSite delivers messages in the form of *message containers*. A message container is first created at the sender process and filled with information. The DMSL transports the container to the target process in a serialized format. At the target process, the serialized message is turned into a message container again. The received message conatainer is passed to the application. Marshaling and unmarshaling of a message containers contents is local to the DMSL instances at the orgin and target processes and not exposed to the application level.
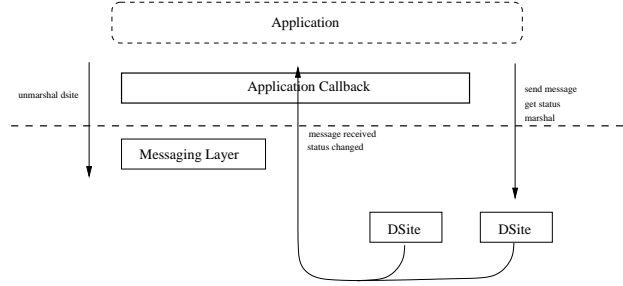
Figure 3: The interface between an application and the DMSL. DSite objects are used for communicating with and reasoning about processes. Messages received are passed to the application over the application callback interface, the DSite representing the sending process are passed as an argument.

The message container implements a queue of abstract items. Data is written to a message container item by item, and read from the container in the same order. The message container can transport sets of items of different types. This allows a message container to be passed between different layers of an application, where each layer can add items to the message container, without any global knowledge of message layout. Naturally, at the receiving process the message must be passed to the same layers as at the sending process, but in the reverse order. In addition, each layer at the receiving process must read every item inserted by the same type of layer at the sending process.

```cpp
class MsgContainer{
public:
  virtual void pushDSiteVal(DSite*) = 0;
  virtual void pushIntVal(const int&) = 0;
  virtual void pushADC(ExtDataContainerInterface*) = 0;
  virtual void pushMsgC(MsgContainer*) = 0;

  virtual DSite* popDSiteVal() = 0;
  virtual int popIntVal() = 0;
  virtual ExtDataContainerInterface* popADC() = 0;
  virtual MsgContainer* popMsgC() = 0;

  virtual bool m_isEmpty() const = 0;
};
```

The message container, the interface is depicted above, can transfer four types of data items. (1) DSite references, (2) integer values, (3) other message containers, (4) opaque data. For example, in the case of the DSS, the internal consistency protocols are completely realized using 1 and 2. The opaque data type, represented by an instance of the `DataContainerInterface` class, is used to transfer programming system specific data that cannot be expressed in the predefined types (this is explained in Section 3.4).

Figure 4 depicts messaging using the DMSL. The message is created at application level and passed to the DSite that represents the target process (1). The message is transferred over the network (2). At the receiving process, process *B*, the message is delivered to the callback interface. Furthermore, the loop-back property of the DSite is also depicted. A message sent at process *A* to the DSite that represents process *A* (4) results in a callback (5) similar to if a message had been received from a remote process. In both cases the messages are passed to the DSites and received over the application callback interface in the format of message containers.
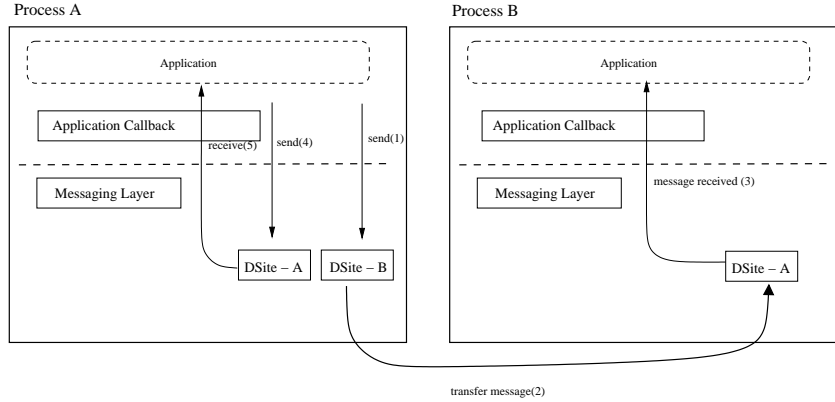
8

Figure 4: The figure depicts communication within one process and between two processes. The DSite representing the local processes is from application level no different from a DSite representing a remote process.

3.3 Sharing Buffers With DMSL

Access to memory buffers allocated by DMSL is provided in the form of buffer interfaces. The buffer size is fixed and buffers are either of read type, or of write type. The interface is shown here:

```
class WriteBuffer{
  virtual void writeToBuffer(const BYTE* ptr, size_t write) = 0;
  virtual int availableSpace() const = 0;
  virtual void         putByte(const BYTE&) = 0;
}

class ReadBuffer{
  virtual int availableData() const = 0;
  virtual void readFromBuffer(BYTE* ptr, size_t wanted) = 0;
  virtual const BYTE    getByte() = 0;
}
```

3.4 Marshaling Data as Late as Possible

DMSL only provides asynchronous messaging. Limitations in the capacity in the underlying I/O facility can make it impossible to transfer the DMSL level messages to remote processes in the same pace as messages are created at Application level. Messages are sent only when the DMSL is get access to the I/O. According to the design of the DMSL, access to I/O is defined by the communication component, see Figure 2.

The DMLS uses a marshaling technique called late marshaling. That is, messages are serialized first when the communication medium can transfer the message. Thus, queued messages in the DMSL are stored as message containers. This is in difference to the commonly used technique, called early marshaling, where a message is passed from application to messaging in a serialized format.

Late marshaling generally uses less buffer space than early marshaling. For example, the memory footprint of a serialized representation of a data structure is commonly larger than the structured format. Furthermore, if the same data is sent multiple times, an early marshaling schema will allocate unnecessary buffer space for each instance of the same data item.

However, late marshaling requires knowledge of how to serialize sent messages inside the messaging layer. This breaks the interface boundary between the DMSL and

9

an application. The DMSL uses an object oriented approach, the message is supposed to know how to serialize its contents. This is true for the DSites, the integer values and the message containers stored as items in a message container. However, nothing is known, at the level of the messaging layer, about the opaque data structures. Opaque data structures are actually instances of the `DataContainerinterface` class. Thus, a container does not only store the data to be sent, but also a description of how to serialize the data. The interface, depicted below, requires implementation of methods for marshaling and unmarshaling.

```
class DataContainerInterface{

public:
  virtual BYTE getType() = 0;
  virtual bool marshal(WriteBuffer *bb, DSite* destination)=0;
  virtual bool unmarshal(ReadBuffer *bb, DSite* source)=0;
  virtual void dispose()=0;
  virtual void resetMarshaling() = 0;
};
```

When marshaling a data container, the marshal interface is called with a write buffer and the destination DSite as argument. The latter can be used for marshaling format optimizations. The constant size of the `WriteBuffer` sometimes requires the contents of a container to be split into sub-parts. Thus, the container is required to be able to interrupt its marshaling (and return **false**). It will later be called to continue marshaling when more space is available in the buffer. Consequently, the unmarshaling interface of a container must be able to express that it has received a fragment of the complete description (and then return **false**).

The DMSL supports multiple types of application level `DataContainerInterface` instances. Each type is identified by a unique byte, returned by the `getType` method. At the receiving process, the type of the container is used to instantiate a container of the right type. The DMSL transports the type and asks the application to instantiate a container.

A message container and its contents are first reclaimed when the message has been successfully delivered. This is automatically taken care of for the internal data structures (DSite, integer, and other message containers), but must be handled explicitly for the opaque `DataContainer`. The `DataContainer` exposes a dispose interface, called when the contents have been successfully transferred. Thus, it is the DMSL that controls the destruction of sent data containers.

3.5 Failure Model

The DMSL classifies the status of a known remote process in one of three states:

No problem. The process is reachable; messages can be sent to and received from the process.

Permanently lost. The process will never be reachable. No messages can be sent to and no messages will be received from the process.

Temporary Lost. The process is unreachable, but it is not possible to correctly classify this as a permanent property. However, the lost status can go away and the DSite changes into the no problem state, or, the process can be detected as lost, and the DSite changes into the permanently lost state.

The state of a DSite object can alter over time to reflect the status of the process it represent. The possible transitions are depicted in Figure 5.
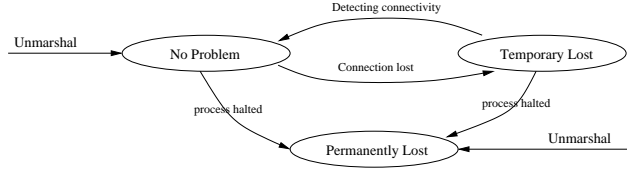
Figure 5: The figure depicts the possible transition between the different states a DSite can be in. Note that an unmarshaled DSite has either *no problem* or *permanent problem* status.

It is the *connection module* that detects and defines when a DSite should do a state transition. Correctly defining permanently lost is known to be hard and for some types of distributed systems impossible. However, it is sometimes possible for a process to learn that a process on the same LAN has terminated. Because of the problem to correctly detect that a process has terminated, halted processes are commonly defined as being temporary lost. In difference to temporary lost that is local to a DSite at one process, permanent lost is a global property. No process can communicate with a process that has halted. Thus, permanently lost information about a DSite is slowly spread among DMSL using a diffusion scheme.

Messages queued for delivery over a DSite that is defined as being permanently lost will never be delivered. The messages are handed back to the application for destruction over the m_unsentMessages method of the AppMslClbkInterface class.

How to detect and classify the different fault states for a given process is different from application to application. To enable simple customization of failure classification the task is lifted out into the connection component (see Figure 2).

3.6 Automatic Resource Management

The DSite realizes a seamless communication channel to the process it represents (modulo failure). This requires connection establishment, detection of lost connection, reconnection at connection loss, and resending of lost messages. The DMSL closes unused connections in order to minimize the inherent cost (in memory buffers, potential file-descriptors, and control messages) of keeping a connection open.

Opening of connections and maintaining connectivity is driven by the existence of messages to deliver. Closing of connections and reclamation of DSite objects is governed by a mark and sweep garbage collection scheme. The set of locally existing DSite objects are swept by invoking the m_gcResource method of the DMSL that schedules wvery unmarked DSite for removal. The DSite object provides an interface for marking application usage of the object. A mark of a DSite object lasts until the next sweep, when the mark is removed. Marking a DSite object multiple times will result in just one mark.

4 The Abstract Entity Layer

The abstract entity layer provides a programming system independent interface that models the entities required for providing transparent distribution on programming sys-
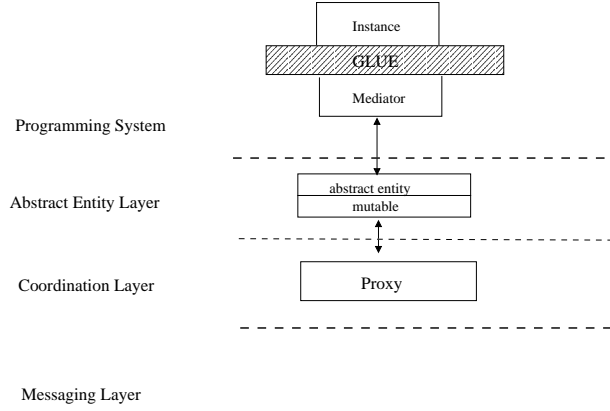
Figure 6: The three item stack of a shared data structure instance. The Instance implements the programming system interface while the proxy implements distributed coordination between all instances that represents a distributed data structure. The abstract entity is the interface between the DSS and programming system items (proxy and instance).

tem level. The prime component of the abstract entity layer is the *abstract entity*.

An abstract entity provides a uniform interface to a large number of eligible protocols. It exposes an interface that allows for access of distributed data structure through a notion of abstract operations. Internally, the abstract entity translates the abstract operation into a protocol operation on the consistency sub-protocol of the proxy the abstract entity is connected to. Furthermore, the abstract entity acts as an interface of the consistency sub-protocol for interacting with the entity instance.

The protocol that ensures a given consistency model for the shared entity is executed over a coordination network. Membership in such a network is represented by the *proxy*. The proxy use the messaging layer for its communication, and it uses the abstract entity to interact with the programming system data structure. The true nature of the programming system is hidden from the proxy behind abstract representations. This includes abstract representations of data structures (abstract entity), threads, operations and operation results.

An instance of a distributed data structure is represented by three entities, the programming system level data structure, the abstract entity and the proxy (see Figure 6). The figure also depicts the layout of the data structure instance. A mediator interface allows the abstract entity to communicate with the instance. How the Mediator interface is connected to the Instance is not specified and is part of the glue that connects the DSS to a programming system. The glue is explicitly depicted by the grey box in the figure.

Three different types of abstract entities are supported by the DSS: mutable, immutable and transient abstract entities. Each abstract entity type is represented by a class. This section describes the classes and interfaces of the abstract entity layer.

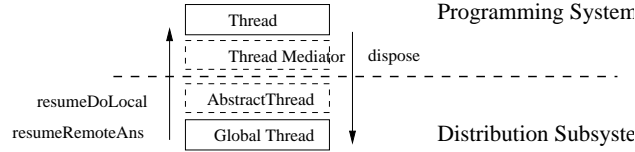|	Thread		Programming System
	Thread Mediator	dispose	
resumeDoLocal	AbstractThread		
resumeRemoteAns	Global Thread		Distribution Subsyste

Figure 7: A programming system level thread made global. The thread is associated with a global thread id that is used to identify the thread when passed between different processes. Thus, one logical thread is potentially represented by multiple programming system threads located at different machines. The figure depicts the interfaces exposed by the DSS and required by the programming system (over the callback class).

4.1 The Program System Term Container

The abstract entity interaface requires transfeing of programming system data. Three types of information are transferred: (i) description of an operation on a programming system data structure, (ii) description of the current state of a data structure, (iii)the results of performing a remote operation. Programming system level information transferred by the DSS is transported in the form of Programming System Term Containers (PSTC). The PSTC is a direct mapping of the `DataContainerInterface` from the DMSL (see section 3.4) and is an interface for the programming system to implement.

```
class PSTC{
public:
  virtual bool unmarshal(ReadBuffer*) = 0;
  virtual bool marshal(WriteBuffer*) = 0;
  virtual void resetMarshaling() = 0;
  virtual void dispose() = 0;
};
```

Similarly to the `DataContainerInterface` the marshaling method must be able to suspend itself in the case of insuficint buffer space. For various reasons a partly marshaled message cab be resent. For that reason the PSTC is required to implement the `resetMarshaling` interface. The programming system is required to implement a method to create a new PSTC, used when unmarshaling a received message.

Internally, the DSS makes use of multiple instance of the `DataContainerInterface`. The PSTC is one of the instances, dedicated to transport programming system level data.

4.2 An Abstract Representation of Threads

Interaction with an abstract entity is based on the notion of logical threads (just thread for short). It is a thread that performs an abstract operation. It is a thread that the abstract entity suspends and later resumes. However, the DSS has no knowledge of a programming system level thread, in some cases it does not even exists a programming system notion of a thread. For the reason of portability the DSS works on an abstract representation of a thread. Every programming system level thread that interacts with shared data structures must have a DSS representation in the form of a *global thread* instance. The association is bi-directional, thus a global thread is associated with the programming system thread, see Figure 7.

The global thread is used to preserve the logical identification of a thread which performs a remote operation. When doing a remote call, a new thread instance will be

created at the programming system level. The new instance is represented by a global thread with the same identity as the global thread of the initiating programming system level thread, depicted in Figure 8.

4.3 The Abstract Operations

The semantics of an operation on a shared data structure is not known at the level of the DSS. Nor is the layout or structure of an operation understood. In order to bridge the gap between the DSS and a programming system operations are translated into abstract operations. An abstract operation should express the same (on an abstract level) semantics as the original programming system level operation. It is the responsibility of the programming system (the glue) to translate operations to appropriate abstract operations.

An abstract operation takes as argument the identity of the thread that executes the language operation and a description of the operation on the data structure. The thread identity is in the form of a global thread. The return value from the abstract operation tells the calling thread how to continue, either perform the operation on the entity instance, or suspend. If the calling thread is asked to suspend, it will later be resumed and either passed the result of the language operation or asked to perform the operation locally. An example of the abstract operation *write* of the mutable abstract entity is shown below. The method returns **true** if the calling thread can perform the operation on the data structure instance, else the thread is suspended:

```
bool abstractOperation_Write(DssThreadId *id,
                            PstOutContainerInterface **& pstout);
```

An abstract operation can be executed locally, without interaction with other processes. Thus, no operation description is passed over the network. The DSS intentionally optimizes this case, and defers creation of the PSTC until an abstract operation call returns. Thus, a PSTS is only created if explicitly needed. The PSTC argument is passed as a reference to a pointer, initialized to NULL. When the call returns, the value of the `pstout` indicates whether a PSTC is required or not. Only if the pstout points to an address, a PSTC should be constructed and assigned the pstout.

4.4 Resolving Programming System Level Operations

For an abstract operation which results in remote execution the original operation must be transported to the remote process. Thus, the operation must be packed into a PSTC that is passed over the network. At the process where the operation is to be resolved, the callback interface for the shared data structure is called by the abstract entity. A callback is passed the global id of the calling thread, a unique operation id, the operation in the form of a PSTC, and a pointer to a possible answer. The DSS has automatically created the global thread identity, but no programming system level thread is yet associated with it. The write callback for the mutable abstract entity is shown here:

```
class MutableCallback{
 bool callback_Write(DssThreadId * id_of_calling_thread,
                    DssOperationId* operation_id,
                    PstInContainerInterface * operation,
                    PstOutContainerInterface *& possible_answer);

};
```

The DSS is single threaded, thus in order to not block further execution, a callback must return as fast as possible. The interface allows for either spawning a thread to resolve the operation (the usual case when doing an RMI) or perform the operation
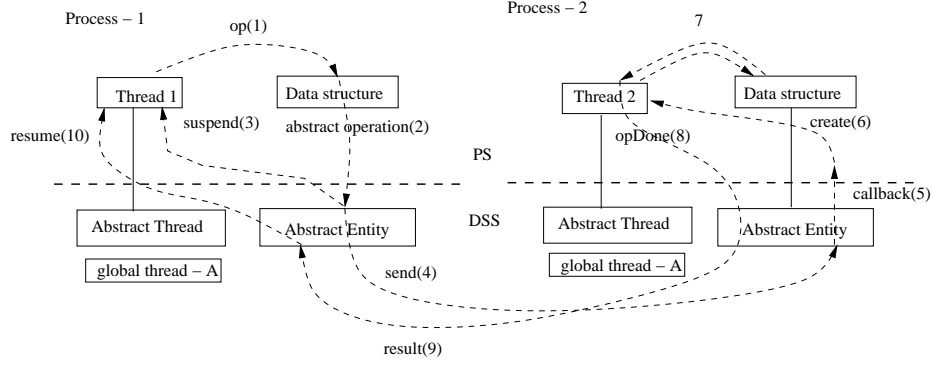
Figure 8: Resolution of a remote operation initiated at $process_1$ and executed on $process_2$. The remote operation is described on the level of data structures, threads and abstract entities. $Thread_1$ initiates the remote operation by performing an operation on the data structure instance. The remote call results in creating a thread at $process_2$ that executes the operation. Note that $thread_2$ has the same global thread as $thread_1$. Thus, conceptually the two thread instances represent the same logical thread.

immediately (an optimization used when the operation is native in respect of the programming system, a typical example would be access of an array). Whether the operation is resolved immediately or not is reflected by the return value (**true** indicates that the operation is completed). If the operation is completed, the `possible_answer` pointer refers a PSTC containing the result of the operation.

The `DssOperation` object is used to identify a non immediate operation and is unique for each callback. Upon completion, the result is passed back to the abstract entity over the `remoteInitatedOperationCompleted` interface (se below). The method takes as argument a PSTC containing the result of the operation and the `DssOperation` that identifies the operation.

```
class AbstractEntity{
  void remoteInitatedOperationCompleted (DssOperationId* id,
                          PstOutContainerInterface* result);
}
```

Figure 8 depicts, on a conceptual level, how a remote operation is resolved. First, $thread_1$, located at $process_1$ performs an operation on a shared data structure (1). Since the data structure is attached to an abstract entity, the operation cannot be resolved by the data structure. Instead the operation is translated into the appropriate abstract operation, that is performed on the abstract entity (2). The original operation on the data structre is passed as as argument to the abstract operation. The consistency protocol of the abstract entity suspends the calling thread (3) and sends the operation to the remote $process_2$ (4). The message contains the global thread identity of the calling thread, and a description of the programming system level operation (as a PSTC).

Upon receiving the message, the abstract entity of $process_2$ is asked to resolve the operation (5). The operation is non immediate, in order to not monopolize the DSS, a dedicated program system thread is created (6) to execute the operation. The thread is initiated with the operation to execute, a reference to the programming system level data structure and the operation id. When the operation on the data structure is finished (7), the operation-id is used to pass the result back to the abstract entity (8).
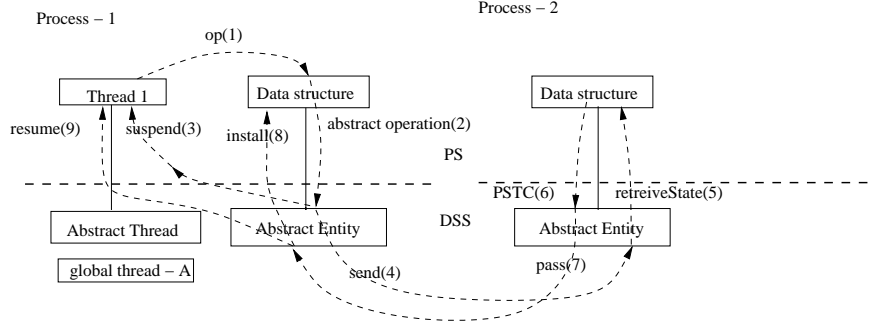
15

Figure 9: The figure depicts transfer of the state description from $process_2$ to $process_1$ in order to let a thread perform a local operation. When the thread at $process_1$ executes the operation on the data structure instance, the instance is in an incomplete state, i.e. *skeleton*. By transferring a state description from $process_2$, the data structture instance is made complete, and the thread can perform the operation locally.

The consistency protocol passes the result over the network back to $process_1$ (9). At $process_1$ the suspended thread is resumed, and handed the result of the operation (10).

4.5 Transferring State

Apart from expressing remote operations the abstract entity allows for local access. This is possible even if the data structure instance is in an inconsistent state when invoked. Local access is provided transferring a correct state description to, and making the local instance conform to the correct state. Naturally, the calling thread must be suspended while the state is transferred. Thus an abstract entity requires the ability to transfer the description of a shared data structure's complete state. The `re-treiveRepresentation` method is used to retrieve a PSTC containing the state description, while the `installRepresentation` is used to install the state to an existing entity instance.

```
class MutableCallback {
  PstOutContainerInterface * retrieveRepresentation ();
  void installRepresentation ( PstInContainerInterface * );
}
```

Note that a state description is not nessesary a incomplete description of the data structure (see Section4.6). A state descritpion is required to contain enough information that a local instance of the same data structure type can be turned into the same state as the instance the description was retrieved from.

In contrast to a remote operation, the act of moving a state description is not related to a programming system level thread. Thus, no thread identity is passed over the network. Since the operation will be executed at the process where it was initiated, nor is the programming system level operation passed.

The interaction between two data structure instances, located at two different processes (process 1 and 2), when the consistency protocol moves the state is depicted in Figure 9. The first three steps are similar to the interaction when passing an operation (see Section 4.4). Instead of sending the operation to $process_2$, the abstract entity sends a request to $process_2$ asking to transfer the state from $process_2$ to $process_1$ (4). The abstract entity is asked by the protocol to retrieve a state description (5). The state

16

description is passed to the abstract entity in a PSTC (6) and passed back to $process_1$ (7). At $process_1$, the state description is installed in the local data structure instance (8), the suspended thread is resumed (9) and allowed to access the data structure instance locally.

4.6 Constructing, Exporting, Importing and Deleting Abstract Entities

A data structure is either *local*, it can only be accessed from one process, or it is *distributed*, it can be accessed simultaneously from multiple processes. The transition of a data structure from being local to being distributed is called *globalization*, and the transition from distributed to local is called *localization*. A distributed data structure is associated with an abstract entity. Thus, globalization is when a data structure is associated with an abstract entity. Localization is when a data structure is no longer connected to an abstract entity.

Globalization is initiated from programming system level and can be initiated at any point in time. Naturally, a data structure must be globalized when a reference is passed to a remote site in order to create a distributed data structure, and not a replica of the data structure at the remote process. Localization, i.e. removing the abstract entity from a data structure, should not be performed without permission from the abstract entity. Removing an abstract entity without permission is equal to creating a local uncoordinated replica of a shared data structure.

4.6.1 Creating an Abstract Entity

Interfaces for creating new abstract entities are provided by the DSS class. The interface takes as argument the choice of consistency-, reference-, and coordination sub-protocol and returns a new abstract entity instance initialized with the chosen sub-protocols. The sub-protocols define the functionality of the proxy and are explained in detail in Section 5. Below is the method of the DSS object that creates a mutable abstract entity.

```
class DSS_Object{
  MutableAbstractEntity *
  m_createMutableAbstractEntity(const ConsistencySP& cnst,
                                const CoordSP& crd,
                                const ReferenceSP& ref);
}
```

The returned abstract entity is not connected to any data structure. For the data structure to be properly globalized, the abstract entity must be associated with the data structure. For the reason of portability, the abstract entity does only communicate with an instance of the `Mediator` class:

```
class AbstractEntity{
  void assignMediator(MediatorInterface *mediator);
};
```

4.6.2 Exporting an Abstract Entity

A reference to a globalized data structure is passed over the network in three sub-parts. First, a programming system level proxy description of the data structure, second, an abstract entity description, and finally a possible programming system level state description. The abstract entity provides an interface for marshaling a description into a `WriteBuffer`. The method takes as argument the target buffer and returns a boolean value, telling whether a state description should be appended or not.

```
class  AbstractEntity{
  bool        marshal(DssWriteBuffer         *buf) = 0;
};
```

Note that the default is to transport programming system level proxy descriptions, and only if the abstract entity decides must a complete description be transported.

4.6.3 Importing an Abstract Entity

After a reference to a globalized data structure is passed from one process, the sender, to another process, the receiver, there exists an instance of the data structure in the address space of the receiving process. If an instance of the distributed data structure did exist in the receiving process address space, no new instance will be created. instead the existing instance will be returned, indicated by the return value of the `unmarshalProxy` method, **true** indicates that the abstract entity (and thus the entity instance) already existed.

```
class  DSS_Object{
  bool    unmarshalProxy(AbstractEntity* &proxy,
                         ReadBuffer* const buf,
                         AbstractEntityName&       cm,
                         bool& trailing_description);
};
```

The definition of the `unmarshalProxy` method is depicted above. The call takes a `ReadBuffer` as argument which should contain the serialized representation of the abstract entity. The type of abstract entity is returned over the `cm` argument, the actual abstract entity is returned over the `proxy` argument. The `trailing_description` argument returns whether a complete description follows or not (similarly to the `AbstractEntity::marshal` method).

When the `unmarshalProxy` method returns an already existing abstract entity, no data structure instance should be constructed at the programming system level. Instead should the data structure instance already associated with the abstract entity be used. The abstract entity provides an interface which returns the Mediator the abstract entity points to:

```
class  AbstractEntity{
  MediatorInterface *accessMediator()
}
```

4.6.4 Removing an Abstract Entity

The presence of a data structure instance at a process indicates that the distributed data structure is referred from the process. If no reference exists to the data structure instance, it should be removed. However, each coordination network potentially maintains a distributed garbage collection algorithm. Thus, a data structure instance and its associated abstract entity cannot be removed without interaction with the DSS.

```
enum  DSS_GC{
  DSS_GC_NONE,
  DSS_GC_WEAK,
  DSS_GC_PRIMARY,
  DSS_GC_LOCALIZE
};

class  AbstractEntity{
  virtual DSS_GC getDssDGCStatus();
  virtual void    clearWeakStatus();
};
```

In order to remove an abstract entity or a data structure instance (this includes the abstract entity), the operation must be permitted by the abstract entity. The abstract

entity has a *root status* that tells the relationship between the abstract entity and the data structure instance. An abstract entity is in one out of four states. First, **none**, the entity instance can safely be removed. Second, **weak**, the state of the instance is of importance for the consistency of the shared data structure and the instance cannot be removed. However, the weak status of the instance can be removed. Third, **primary**, the instance cannot be removed since it is used to uphold the consistency of the distributed entity. Last, **localize**, the abstract entity can be removed, and the instance can be made a local data structure.

The `clearWeakStatus` method is used to initiate clearing of the weak status. The proxy will then try to remove, if possible, the information that makes it a root. If removed, the abstract entity will have root status **none**. The abstract entity does not signal the glue when the **weak** status is removed. The programming system is assumed to periodically ask the abstract entity about its current state.

Deleting an abstract entity that reports **primary** or **weak** root status will prevent further access to the shared data structure the abstract entity controls, i.e. no proxy can access the shared data structure. Removing an abstract entity in the **none** root status and allowing further access to the local instance is similar to making an un-coordinated, local copy of the shared data structure.

5 Components of the Coordination Network

Each distributed data structure is controled by a consistency protocol, called a distribution strategy. Every node that holds a reference to a distributed data structure maintains an abstract entity, and executes the associated distribution strategy. The nodes that executes a particular distribution strategy, forms a virtual network, called the coordination network. The coordination network consist out of a set of proxies and a coordinator. Each proxy is associated with one of the abstract entity instances that represents the distributed data structure. The coordinator is similar to the home in a home-based protocol. It is assigned arbitrating tasks, suchs as keeping network references to where the current state of a mobile object is currently located.

The entity consistency protocol executed over the coordination network is divided in three sub-protocols. Each sub-protocol type is represented as a component. Sub-protocol instances of the three different types can be freely combined to form customized consistency protocols. The *consistency* sub-protocol realizes access of a distributed data structure. The *reference* sub-protocol is responsible for detecting when the coordination network can be dismantled. The *coordination* sub-protocol implements a home-based communications infrastructure over the coordination network, i.e. a communication service that allows a proxy to send a message to its coordinator.

Every sub-protocol is realized by two types of instances. First, a home-instance, located at the coordinator. Second, a remote-instance, an instance is located at each proxy. Communication is contained to the sub-protocol type, no messages are sent between different sub-protocols of a coordination network, thus the coordination sub-protocol does not send a message to the consistency sub-protocol. This allows the sub-protocols to privately define their own message types and message formats. Consequently, adding a new instance of a sub-protocol type to the DSS only requires extending the sub-protocol creation primitives, which is a minor change to the system.
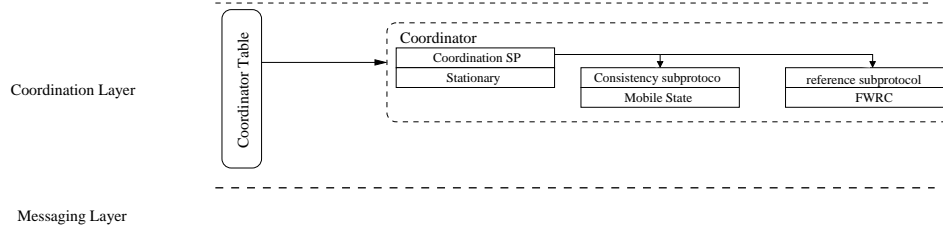
Figure 10: The layout of the coordinator, its internals and the coordinator table. Locally at each process there exists a coordination table that is, given an coordination network identity, used to find a coordinator. Internally, the coordinator is represented by three objects that implement the three sub-protocols. This coordinator is configured with a stationary coordination sub-protocol, a mobile-state consistency sub-protocol, and a weighted reference counting reference sub-protocol.

5.1 The Coordination Network

The coordination network has one purpose, to maintain consistent access to a shared data structure. Here the components of the coordination network are introduced.

5.1.1 The Coordinator

The coordinator is created at the process where the coordination network is initialized. Depending on the type of coordination sub-protocol, the coordinator is either fixed to its creation process or can move between processes. Disregarding how the coordinator behaves, it is of outmost importance that the proxies can find the coordinator.

A coordinator is represented by a coordinator instance object. Internally, the coordinator is represented by three different objects, one for each sub-protocol (see Figure 10).

5.1.2 The Proxy Object

In order to get access to a consistency protocol, a proxy object is required. The proxy object is logically located in the coordination layer of the DSS and connected to an abstract entity. Internally, the proxy is represented as a set of objects. The conceptual proxy object, the instance referred by the abstract entity, is the same as the coordination sub-protocol object. The other two sub-protocols, for reference and for consistency, are represented as separate objects, referred to by the coordination object. The layout of the proxy is depicted in Figure 11.

In order to join a corodination network a proper proxy is required. A proxy can only be created from a serialized representation. For a given coordination network, only proxies of the coordination network can produce proper serialized representation.

At any single process, there can at most be one proxy-instance per coordination network. This is automatically controlled by the Proxy table, depicted in Figure 11). Reception of a proxy description will either result in creation of a new proxy or the use of an already existing proxy. A proxy description contains information that in some cases is state-full in the meaning that it cannot just be thrown away without loosing
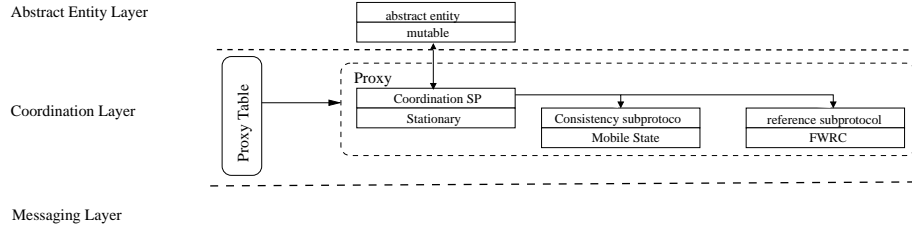
20

Figure 11: The layout of a proxy instance. The proxy is member of a coordination network is configured with a stationary coordination sub-protocol, a mobile-state consistency sub-protocol and uses fractional weighted reference counting as garbage collecting algorithm. This is shown by the specialization of the different sub-protocol objects.

data necessary for the correctness of some of the coordination network sub-protocols[3]. Internally, if an serialized description of an already existing proxy is received, the received information is desterilized by the existing proxy. This is called a *merge*.

5.1.3 Inter Coordination Network Communication

Every coordination network has a globally unique name. The name is used for addressing components and for avoiding duplication of proxies. When a process receives a serialized representation of a proxy it first unmarshal the name of the coordination network (found first in the serialized representation). The name is used to check if an instance already exists or not. If an instance exists, the instance is asked to discard the serialized representation.

The coordination layer maintains two tables, one for proxies and one for coordinators. The tables, depicted on the left in Figure 10 called Coordinator Table and on the left in Figure 11 and called Proxy Table, allows for mapping globally unique names to coordinators and proxies respectively.

A message sent within a coordination network is addressed to a proxy or a coordinator at a particular process. Upon reception of the message, the message is passed from the messaging layer to the dispatcher. The dispatcher reads the message type and the target address and hands the message to the target proxy or coordinator.

The destination type is further refined and tells the type of the sender. The resulting four message types are proxy-to-proxy, proxy-to-coordinator, coordinator-to-proxy and coordinator-to-coordinator. The information regarding the sender type is used if the target does not exist at the process. In such case, the message is passed back to the sender process and handed the proxy or coordinator that sent the message. Since the sender type is explicitly known, the message body does not have to be interpreted.

5.2 Sub-protocol Interaction

In order to cater for code reuse and simple customization, the coordination network is implemented by three sub-protocols. Each sub-protocol is realized by two components, a remote instance present at each proxy, and a home instance locate at the coordinator.

---

[3]This is especially true for distributed garbage collection algorithms, a lost message can prevent the coordination network from ever dismantling itself, even when no remote references exists.
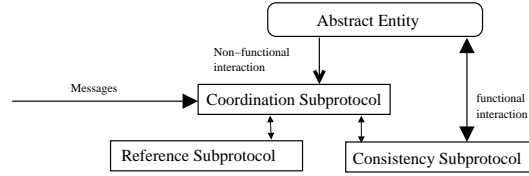
Figure 12: On the level of interfaces the proxy is different from the coordinator in that it communicates with the abstract entity. This figure depicts the interaction regarding operation resolving betwen the coordination sub-protocol and the abstract entity. The coordination sub-protocol exposes an interface that allows for control of the coordination-network (e.g. explicit migration of the coordinator).

Each component of the sub-protocols is represented by an abstract class. The different sub-protocols interact internally over well defined interafces. Moreover, the remote-instances of the different sub-protocols interact with the abstract entity, depicted in Figure 12.

5.2.1 Reference Sub-protocol

The purpose of the reference sub-protocol is to detect when there is just one proxy, and thus the coordination network can be dismantled resulting in localization of the data structure instance. The purpose of the home-instance is to detect when the number of remote-instance reaches zero. Note that both created instances and instances travelling the network in the form of serialized representations must be taken into consideration. The home-instance prevents dismantling of the coordination network as long as the number of remote-instance is non-zero, i.e. the home-instance is a root for garbage collection. For the reason of simplicity, the home-instance is passive; the root status must be retrieved from it. The interface of the home-instance is shown below:

```
class ReferenceSP_home{
  virtual void m_msgRecevied(MsgContainer *msg, DSite* from) = 0;
  virtual bool m_isRoot() = 0;
  virtual void m_makeGCpreps() = 0;

  // Only used by a Proxy coolocated with a coordinator
  virtual void m_getReferenceInfo(MarshalBuffer*);
  virtual void m_mergeReferenceInfo(UnmarshalBuffer*);
};

class ReferenceSP_remote{
  virtual void m_getReferenceInfo(MarshalBuffer*);
  virtual void m_mergeReferenceInfo(UnmarshalBuffer*);

  // control interface
  virtual void m_msgRecevied(MsgContainer *msg, DSite* sender) = 0;
  virtual void m_makeGCpreps() = 0;


  virtual bool m_isRoot() = 0;
  virtual void m_makePersistent() = 0;
  virtual bool m_drop() = 0;
}
```

The common property of the eligible distributed garbage collection algorithms is that they can tell whether the number of outstanding references is zero or more. This puts an requirement on a proxy that is collocated with the coordinator. If it would retain a remote instance, the coordination network would never be dismantled. To avoid this deadlock, the proxy collocated with the coordinator share the home-instance of the reference sub-protocol with the coordinator. Consequently, it has no remote-instance, and is not accounted for when it comes to the distributed garbage collection (see Figure
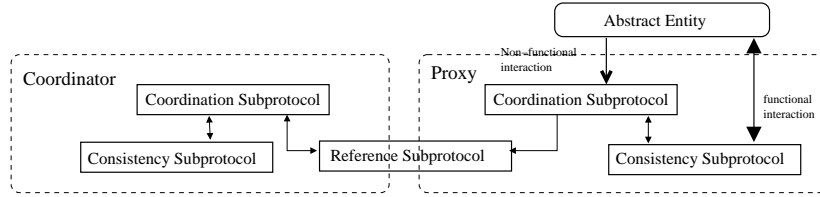
22

Figure 13: A proxy that is collocated with the coordinator of the coordination network both belongs to share reference sub-protocol instance. The sub-protocol is belongs to the coordinator, thus the bidirectional arrow between the coordination sub-protocol of the coordinator.

13).

A reference to a proxy passed over the network is attached a remote-instance description, retrieved using the m_getReferenceInfo method. Similarly to replication of proxies, reception of a description will either result in creation of a new instance or merge of the received information into an existing instance. The later is realized over using the m_mergeRefenceInfo method.

At creation of a coordination network, there are no remote proxies and thus the coordination network is subject to be dismantled (if the reference sub-protocol can detect this). However, as mentioned above, the remote instance of the reference sub-protocol is passive. Eventual dismantling is initiated from the programming system level (see Section 5.2.4).

5.2.2 Consistency Sub-protocol

The consistency sub-protocol has two roles. Primary, it is responsible for maintaining consistency invocations of the different local data structure instances that represents the distributed data structure. This is realized by interaction with the abstract entity the proxy is connected to, described in detail in Section 5.4. The relationship between the consistency sub-protocol and the abstract entity is depicted by the arrow that connects the two components in Figure 11. Moreover, it implements interfaces for interacting with the coordination sub-protocol.

The home instance implements an interface for receiving messages, and an interface for migrating the home instance. The later method is used when the coordinator is migrated[4]. At migration, the state of the home instance should be packed in the Msg-Container msg. At the process to which the coordinator moves, the home instance will be recreated using the information packed in the msg. The interface is depicted below:

```
class ConsistencySP_home{
  virtual void m_msgReceived(MsgContainer* msg, Dsite* sender) = 0;
  virtual void m_migrate(MsgContainer *msg);
};
```

The interface required for the remote instance is more extensive. Similarly to the remote instance, the remote instance must be able to receive messages. A consistency sub-protocol remote instance can be a root for the local garbage collection (see Section 5.2.4) The m_isWeakRoot interface returns potential weak root status. The remote instance should try to remove any weak status if the m_clearWeakRoot method is called.

---

[4]Migration of the coordinator is initiated and controlled by the coordination sub-protocol

23

When a reference to a proxy is marshaled, the consistency sub-protocol is asked to add information to the serialized representation by the marshal_protocol_info method. The consistency sub-protocol defines what type of proxy that should be marshaled. The return value from the marshal method tells whether a complete description of the associated programming system data strutcure should be marshaled, or if just a proxy description should be marshaled.

```
class ConsistencySP_remote{
  virtual void m_msgReceived(MsgContainer* msg, Dsite * sender) = 0;
  virtual bool m_isWeakRoot() = 0;
  virtual void m_clearWeakRoot() = 0;
  virtual bool marshal_protocol_info(DssWriteBuffer *buf, DSite *) = 0;
  virtual bool dispose_protocol_info(DssReadBuffer *buf) = 0:
};
```

5.2.3   Coordination Sub-protocol

The coordination sub-protocol provides the messaging service for the reference and consistency sub-protocols. Furthermore, the remote instance exposes an interface towards the abstract entity that allows for instrumentation the coordination network, depicted by the *non-functional interface* arrow in Figure 12. Similarly to the other sub-protocols the coordination sub-protocol is defined as an interface that can be instantiated to implement new behaviors. The natural examples are stationary and mobile coordination sub-protocols, both currently supported by the DSS.

A messaging interface is provided for the reference and the consistency sub-protocols. A message is typed with the destination, proxy or coordinator, and the sub-protocol type. As can be seen in the class definition below, m_createProxyConsMsg creates a consistency sub-protocol message addressed to a proxy, and m_createCoordRefMsg creates a reference sub-protocol message addressed to a coordinator.

```
class CoordinationSP_home{
  virtual bool m_sendToProxy(DSite * dest, MsgContainer* msg) = 0;

  // Only for the ConsistencySP
  virtual :: MsgContainer * m_createProxyConsMsg ()= 0;

  // Only for the ReferenceSP
  virtual :: MsgContainer * m_createProxyRefMsg() = 0;
};

class CoordinationSP_remote{
  virtual bool   m_sendToCoordinator(:: MsgContainer* msg)= 0;
  virtual bool   m_sendToProxy(DSite * dest, :: MsgContainer* msg);

  // Only for the ConsistencySP
  virtual MsgContainer * m_createCoordConsMsg ();
  virtual MsgContainer * m_createProxyConsMsg ();

  // Only for the ReferenceSP
  virtual MsgContainer * m_createCoordRefMsg ();
  virtual MsgContainer * m_createProxyRefMsg ();
};
```

The m_sendToProxy passes message msg to a proxy at process dest. Since there is only one coordinator in the coordination network, the m_sendToCoordinator method takes no explicit destination.

5.2.4   Calculating Root Status of a Proxy

The root status of an abstract entity (see Section 4.6.4) is calculated by the proxy the abstract entity is connected to. If the proxy is collocated with the coordinator, the proxy will ask the coordinator for its root status. Otherwise, the root status of a proxy is resolved by the coordination sub-protocol. The root status of a particular proxy considers the status, first of the reference sub-protocol, second the coordination sub-protocol and

finally the consistency sub-protocol. Below is an example of how the garbage collection status is calculated for the proxy of the stationary coordination strategy:

```
DSS_GC
ProxyStationary :: getDssDGCStatus (){
    if ( a_man == NULL){
        if ( a_referenceSP->m_isRoot()) return DSS_GC_PRIMARY;
        if ( a_consistencySP->isWeakRoot()) return DSS_GC_WEAK;
        return DSS_GC_NONE;
    }
    return a_man->m_getDssDGCStatus ();
}
```

As can be seen above, a proxy collocated with the coordinator asks the coordinator for root status. A proxy that is not collocated with the coordinator checks the reference and consistency sub-protocols for their root status. If none of the two sub-protocols have any root status, the proxy has root status none.

Localization, for the stationary coordination sub-protocol, can only happen to the proxy that is collocated with the coordinator. Below is the code for calculating the root status of the stationary coordinator. Note that if the reference sub-protocol is not a root, the coordination network is subject to localizion.

```
DSS_GC
CoordinatorStationary :: m_getDssDGCStatus (){
    if ( a_homeRef->m_isRoot()) return DSS_GC_PRIMARY;
    return DSS_GC_LOCALIZE ;
}
```

5.2.5 Marshaling and Unmarshaling a Proxy

The proxy provides an interface for writing a marshaled representation of itself to a buffer. The DSS implements routines to construct a proxy from a marshaled description.

The marshaling representation of a proxy contains enough information, in a serialized format, to create and instantiate a proxy at another process. Instantiate includes connecting the proxy to the coordination network and make it functional, i.e. make all sub-protocols able to execute their protocols. The code below depicts marshaling on a somewhat conceptual level. In reality the marshaling is more complicated because of buffer space saving optimizations:

```
void marshalProxy (Proxy * pReadBuffer * buf){
    marshalNetIdentity(p->m_getNetIdentity(), buf);
    marshalInteger(p->m_getCoordinationSP_Type ());
    marshalInteger(p->m_getReferenceSP_Type );
    marshalInteger(p->m_getConsistencySP_Type );
    p->masrhalSubProtocols(buf);
}
```

The sub-protocol types are expressed using numbers. Sub-protocol information is serialized, and is at the receiving process unmarshaled by an instance of the same sub-protocol type.

Creating a Proxy from a marshaled description is the inverse of creating a marshaled representation. The code for unmarshaling of a proxy is shown below:

```
Proxy * unmarshalProxy (ReadBuffer * buf){
    NetIdentity ni          = unmarshalNetIdentity (buf);
    CoordinationSP_Type coordt = unmarshalInteger (buf);
    ReferenceSP_Type    reft = unmarshalInteger (buf);
    ConsistencySP_Type  const = unmarshalInteger (buf);

    // returns NULL if no proxy exists.
    Proxy p = proxyTable->findProxy( ni );
    if (p == NULL){
        // While instantiating the different sub-protocols the
        // sub-protocols can read unmarshal information from the buffer.
        p = createProxy (ni , coord , reft , const , buf);
    }else{
        // The sub-strategies unmarshals the information found in the
        // buffer, and if nessesary make use of it.
        p->discardInformation(buf);
    }
    return p ;
}
```

Note that if a proxy registered under the received global identity exists, that instance is used and no new instance is created. The existing instance is responsible for reading the marshaled data that describes the sub-protocols. This allows the sub-protocol instances to make use of information found in the marshaled description of the sub-protocol component. For example, a marshaled representation of the reference sub-protocol can contain information with token status. Furthermore, in the case of the mobile-coordinator coordination sub-protocol the marshaled instance is used to distribute knowledge of the current coordination location.

5.3 Handling Node Failures

Node and link failures to the nodes of the coordination network will potentially affect its functionality. In the worst case, the coordinator is lost, or information critical to the consistency sub-protocol is lost. In both cases the coordination network is unable to provide services. The coordination framework reports failures to higher levels, i.e. the abstract entity.

Consequently, failure recovery is not required of the sub-protocol instances. Instead, a sub-protocol is required to deduce if a node failure, temporary or permanent, affects the functionality of the sub-protocol.

5.3.1 Reporting Failures

Failures to the coordination network are reported to the abstract entity that in turn reports the failures to the programming system. The coordination sub-protocol implements an interface that returns the current fault state, experienced by the proxy:

```
class CoordinationSP_Remote{
  virtual FaultState  getFaultState();
}
```

A fault state describes if the coordination network provides its service or not:

```
enum FaultState{
  FS_NONE,
  FS_TEMP,
  FS_PERM
};
```

The status of a coordination network is described using the same model as remote nodes (see Section 3.5). FS_NONE indicates that the proxy functions normally. A coordination network that has experienced a fatal error is in the state FS_PERM. A proxy that reports FS_TEMP cannot provide service, however, the problem can go away, thus the proxy then becomes FS_NONE.

5.3.2 Classifying Failures

The messaging layer detects and classifies failures. A state change to a DSite is reported to the coordination layer. In the coordination layer, every proxy and coordinator is informed about the changed state. It is then the task of the proxies and coordinators to deduce if the state change of the particular DSite affects their functionality. Internally, both the coordination sub-protocol and the consistency sub-protocol can be affected. Thus both sub-protocols require implementation of the m_siteStateChange method. Below is remote-instance interface of the coordination sub-strategy depicted:

```
class CoordinationSP_remote{
    void m_siteStateChange(DSite*s, const DSiteState& state)
```

The coordination sub-protocol is responsible for implementing the fault reporting interface. Thus, it is responsible for calculating the complete fault state for the coordination network.

```
void
CoordinationSP_Remote_Stationary :: m_siteStateChange (DSite * site ,
                                              const  DSiteState & state )
{
  FaulState  fs  = FS_NONE;

   // is the affected site equal to the coordinator location
  if (s == m_getGUIdSite ()){
    switch (state){
    case DSite_OK :
      fs = FS_NONE;
      break ;
    case DSite_TMP :
      fs = FS_TEMP;
      break ;
    case DSite_PRM :
      fs =  FS_PERM );
      break ;
    }
  }
  fs = max( fs  , a_consProt->siteStateChanged (s ,  state ));
  setFaultState ( fs );
  a_AbsEnt_Interface->reportFaultState ( fs );
}
```

Above is the code for the m_siteStateChange method of the stationary coordination sub-protocol. Note that the fault status from the consistency sub-protocol is merged with the fault status of the coordination network.

5.4   Interaction Between the Proxy and the Abstract Entity

The functional interaction with an abstract entity and a data structure instance is directed to the consistency sub-protocol of the proxy. The abstract entity merely acts as an interface between the data structure instance and the proxy.

The interface provided by the abstract entity to the consistency protocol is shown below. Note that the callback method is used for all abstract operations. The performed abstract operation is passed as an argument.

```
class AbstractEntity{
  virtual bool callback(int abstract_operation ,
                        DssThreadId * id_of_calling_thread ,
                        DssOperationId* operation_id ,
                        PstInContainerInterface * operation ,
                        PstOutContainerInterface *& possible_answer );
  virtual PstOutContainerInterface * retrieveState () = 0;
  virtual void installState (PstInContainerInterface * builder ) = 0;
}
```

The consistency sub-protocol must implement the two operations handling continuous operations management. Furthermore, it must implement at least one method that allows for interaction with the protocol. Below is a interface provided by the consistency sub-protocol towards the abstract entity . Note that the type of abstract operation is passed as an argument to the operation. Multiple abstract operations can be mapped to the same operation.

```
class ConsistencySP_Remote{
public :
    OpRetVal <OP>(int abstract_operation ,
                  GlobalThread * const ,
                  PstOutContainerInterface ** &);

    void
    remoteInitatedOperationCompleted (DssOperationId* opId ,
                                      PstOutContainerInterface * pstOut );
    void localInitatedOperationCompleted ();
}
```

6 Concluding Remarks

This description of the DSS focuses on key concepts in order to help understanding how the middlewre is actually implemented. This document together with the material published in various conferences about the DSS should give a complete (or near to complete) picture of the DSS. We believe it will server as a useful resource of information for a system integrator that uses the DSS library to create a distributed programming system.

The code-base that makes up the DSS has mainly been developed by Erik Klintskog, Zacharias El Banna with help from Per Sahlin, and Valentin Messaros. Some inheritance on the level of ideas can be traced back to the distribution support for the Mozart system developed by Erik Klintskog, Per Brand, Anna Neiderud, Andreas Sundström and Konstantin Popov. This document would never have been possible without the joint effort invested by the persons mentioned above in design and development of the two systems, DSS and Mozart.