Application Interface to the DSS Middleware Component

Erik Klintskog    Zacharias El Banna

January 13, 2003
## Contents

1. **Introduction** ................................................. 3
   1.1 The DSS Package ........................................ 3
   1.2 The DSS Source ........................................ 3
   1.3 DSS Configurations ...................................... 3
   1.4 The Package Tree ....................................... 4
   1.5 Outline ................................................ 4

2. **The Coordination Assistant - Entity Mediator Pair** .... 5
   2.1 Class CAInterface ...................................... 5
       2.1.1 assignMediator .................................... 5
       2.1.2 accessMediator .................................... 6
       2.1.3 doAbstractOperation - (ordinary) ............... 6
       2.1.4 doAbstractOperation - relation .................. 7
       2.1.5 manipulateCN ....................................... 7
       2.1.6 instrumentMM ....................................... 8
       2.1.7 getDssDG CSStatus ................................ 8
       2.1.8 clear WeakRoot .................................... 9
       2.1.9 dispose ............................................. 10
       2.1.10 getFaultState .................................... 10
       2.1.11 setFaultState .................................... 11
       2.1.12 setRegisteredFS .................................. 11
       2.1.13 getRegisteredFS .................................. 11
       2.1.14 marshal ............................................ 11
   2.2 unmarshalCA .............................................. 12
   2.3 createCoordinationSet .................................. 13
   2.4 instrumentDssMM ........................................ 13
   2.5 Class EMInterface ...................................... 14
       2.5.1 doAbstractOperation ................................ 14
       2.5.2 resumeProceduralThread ................................ 14
       2.5.3 resumeFunctionalThread ............................. 15
       2.5.4 resumeAtomicThread ................................ 16
       2.5.5 reportFaultState ................................... 16

3. **Memory Buffers** ........................................... 16
   3.1 Class DssWriteBuffer ................................... 17
       3.1.1 availableSpace ...................................... 17
       3.1.2 putByte ............................................. 17
       3.1.3 copyToBuffer ....................................... 18
   3.2 Class DssReadBuffer .................................... 18
       3.2.1 availableData ....................................... 18
       3.2.2 commitRead .......................................... 19
       3.2.3 getByte .............................................. 19
       3.2.4 copySafeFromBuffer ................................ 19
4 Message Mediators

4.1 Class MMOutInterface ........................................ 20
   4.1.1 marshal .................................................... 20
   4.1.2 resetMarshaling .......................................... 21
   4.1.3 dispose .................................................... 21
   4.1.4 loopBack2In .............................................. 22
4.2 Class MMInInterface ........................................... 22
   4.2.1 unmarshal ................................................ 22
   4.2.2 dispose .................................................... 22
   4.2.3 loopBack2Out ............................................. 23
4.3 createMMInContainer .......................................... 23

5 The I/O Service .................................................. 23

5.1 Class CallbackHandler ....................................... 23
   5.1.1 invoke ..................................................... 24
5.2 Class DesTransportChannel ................................. 24
   5.2.1 readData .................................................. 24
   5.2.2 writeData ................................................ 25
   5.2.3 closeChannel ............................................. 25
   5.2.4 registerWriteHandler ................................. 25
   5.2.5 registerReadHandler .................................... 25
   5.2.6 unregisterRead .......................................... 25
   5.2.7 unregisterWrite ........................................ 25

6 Representing Remote Processes ................................ 26

6.1 Transferring Site Representations ....................... 26
6.2 Connection Establishment .................................. 26
   6.2.1 Grants and Different Types of Channels .......... 27
   6.2.2 Active Connection ...................................... 27
   6.2.3 Passive Connection ..................................... 28
   6.2.4 Simultaneous Connections ............................ 29
   6.2.5 Sites as Resources ..................................... 30
6.3 Class Site_Name ............................................... 30
   6.3.1 channelEstablished ..................................... 30
   6.3.2 connectionFailed ........................................ 30
   6.3.3 siteFault ............................................... 30
   6.3.4 registerFaultInterest .................................. 31
   6.3.5 getFaultState .......................................... 31
6.4 Class Site_Address .......................................... 31
   6.4.1 establishConnection .................................... 32
   6.4.2 abortConnection ........................................ 32
   6.4.3 grantAquired ............................................ 32
   6.4.4 faultCallback ........................................... 32
   6.4.5 dispose .................................................... 32
   6.4.6 marshal .................................................... 32
   6.4.7 stringrep ................................................ 33
6.5  unmarshalSiteRepresentative ................................. 33
6.6  disposeSiteRepresentative ................................. 33
6.7  gcDssResources ............................................. 33
6.8  Class ConnectionGrant ....................................... 34
   6.8.1  getConnectionType ........................................ 34
6.9  getConnectionSynch ........................................... 34
6.10 getConGrantAsync ............................................ 34
6.11 anonymousChannelEstablished ............................... 35
6.12 freeGrant ..................................................... 35

7  Auxiliary  ..................................................... 35
   7.1  initDSS ....................................................... 36
   7.2  heartBeat ..................................................... 36
   7.3  operateIntParam .............................................. 36
   7.4  operateStrParam .............................................. 37
   7.5  GL_error ..................................................... 37
   7.6  GL_warning .................................................. 38
1 Introduction

The Distribution Subsystem (DSS) is a powerful middleware for distributed programming systems (DPS). The DSS is intended to be coupled with an Extended Virtual Machine that implements a programming language. It is in its design highly modular, and tunable, and offers the means for full transparent distribution of programming languages. The implementation is operating system independent, since all functionality that requires contact with the os is lifted out and implemented by separate services. The design, and the philosophy behind it can be found in the document *A Generic Middleware for Dynamic Intra Language Transparent Distribution*. This document describes the interface to the DSS, and the steps necessary to link an application to the DSS. It is thus advised to first have read the description of the DSS before reading this document.

1.1 The DSS Package

This document describes the object oriented interface to the DSS. The interface is realized with objects. Services provided by the DSS are represented as classes, and services that has to be implemented for the DSS are also represented as classes. The DSS is implemented in C++.

Enclosed with the DSS software package is several components organized in a directory structure. Important to note here is that there are two ways the DSS (the automata) can be handled by an application: either as a *DSS Object* where all the procedures are methods and initialization is the same as object construction. The DSS can also be seen as a regular automata with procedures and an initialization procedure.

1.2 The DSS Source

The complete source package can be downloaded from http://dss.sics.se/, it contains the latest version of the DSS source code and all the additions and components described in this document.

1.3 DSS Configurations

The root directory contains some text files with information regarding how to build the DSS and also the status of the ongoing work with the DSS. Also in the root directory is configure-files for building the DSS in a Linux/gcc environment as well as .Net framework project files for building the DSS in Windows/Visual Studio environment. When building the DSS under Linux/gcc the application programmer can, during configuration, choose to create either an archive file, *libDSS.a*, which can be statically linked to the targeted application, or a dynamically linked library file, *libDSS.so*.

The configuration (Linux) of the DSS-build can be instrumented to also build and link the examples and utilities enclosed in the package. More on how
to do this is explained in the text files located in the DSS-root directory.

1.4 The Package Tree

The subdirectories of the software package, containing the different components is organized as follows:

(src) The source code directory. Contains all the C++ files to build a complete DSS object-file. The object file can after compilation be linked statically or dynamically to the targeted application.

(include) This directory contains the header files with declarations which are exposed to the application the DSS is to be linked to. If the former object approach is chosen the header file the application should include is the *dss_object.hh* and in the other case the *dss_interface.hh* should be included. The other files in this directory contains the enumerators for return values and arguments as well as class declarations for the classes presented in the rest of this document.

(examples) Example files containing various distributed operations. The intention is to show how distributed entities or mechanisms can be implemented as well as for testing of the DSS functionality. The files can of course be compiled into stand alone binaries.

(utilities) The utility directory contains example implementations of an I/O server, base64 encoding/decoding utility and an example of a mediating layer of an EVM, used by the examples.

(documentation) Contains documentation files, like this one.

(dotnet) This directory is the root source when coupling the DSS to the .Net platform (i.e. C#). It also includes some tests. More on how to use the .Net files is described in the directory.

1.5 Outline

The interface to the DSS is organized in sections, where each section describes a functional component. Each functional component describes eventual interfaces provided by the DSS, and services that the EVM should implement. Besides that, each functional component is given a short introduction. Section 2 describes the coordination assistant. All aspects regarding are addressed here, including operational and garbage collection interface. Sections 3 and 4 describes the interfaces for marshaling. The functionality, and the interfaces for the OS-service module are described in sections 5 and 6. The last section, section 7 describes auxiliary interfaces as debug logging and system initialization interfaces.
2 The Coordination Assistant - Entity Mediator Pair

The realization of distributed entities in an EVM is implemented as a pair of two tightly coupled interface objects, representing the entity instance, described in section???. The object on the DSS level is an instance of the CA_Interface class; its methods offer the complete set of operations that can be performed on a Coordination Assistant (CA), including tasks as performing abstract operations (AOP), memory management and fault handling. These methods are a part of the exposed DSS interface and thus supplied to the programmer connecting the DSS to an EVM.

The Entity Mediator, or EM_Interface, is the object located in the mediation layer of the EVM, representing the Local Representative (LR) for the CA. It offers the callback interfaces the CA needs to do operations on the entity. These interfaces are generalized (into atomic, functional and procedural), thus the real operations are to be implemented by the EVM, translating the abstract entity-depending calls into real VM operations on the local representative. Note that all the interface functions of this class (the Entity Mediator) must be implemented by the programmer connecting the DSS to an EVM.

Instances of the CA_Interface are created explicit by the EVM, during globalization when the EVM detects that an entity should be shared. The EVM invokes a special create-coordination-set method of the DSS, returning a CA_Interface instance which then is tied to the Entity Mediator explicitly. Another case of creation is when a entity instance is unmarshaled from a Message Mediator, the unmarshaling method will notify the EVM if it should assign a new handle to the returned CA_Interface, or if this CA_Interface already existed.

Note that the DSS uses two Message Mediators with different purposes, one for outgoing messages that should be marshaled and one for incoming messages which are unmarshaled. These are explained later so for now they are only referred to as MMIn_Interface and MMOut_Interface.

2.1 Class CA_Interface

Each coordinator assistant is reachable through their respective interface objects and all operations are available through their methods. The coordinator assistant works in conjunction with an entity mediator. The entity mediator is the end point the coordinator assistant uses to perform callbacks on. The methods offered by the CA_Interface are all implemented into the DSS and are described here:

2.1.1 assignMediator

```c
void assignMediator(EM_Interface* mediator)
```

The method handles the assignment of an EM to the CA and should be invoked directly after the DSS returns a new handle to a CA (either through unmarshal-
ing, see 2.2, or explicit creation, see 2.3). This is not a once only operations so
if the EVM wants to change the associated EM, or just repoint it, it may do so
and the effect will be noticed by the DSS immediately after invoked.

The method takes a pointer to a EM_Interface which is the representation
of a EM.

2.1.2 accessMediator

EM_Interface *accessMediator(void)

The method returns a pointer to the currently assigned entity mediator.

2.1.3 doAbstractOperation - (ordinary)

OpRetVal doAbstractOperation(const AbsOp& aop,
const DssThreadId& id,
MIOut_Interface* mmout)

The Abstract Operation on a CA can be performed using one out of two meth-
ods. Either an operation can be performed on a distributed entity, the normal
case, or the entity can be associated with another entity, this is referred to as a
relation operation. This ordinary method refers to the operation on an entity.

To perform an abstract entity operation on a distributed entity this method
is invoked. It will pass the abstract operation, a caller id and the argument to
the consistency protocol and the protocol will instruct the calling unit on how
to proceed. The has taken control of the argument (mmout).

Performing the abstract operation will instantaneous complete and return a
result, hence, the call will not block the process. However, the return value can
instruct the original calling thread to suspend itself.

AbsOp The abstract operation to be performed on the entity. The operation
must be defined in the set of possible operations defined by the abstract
entity chosen for the entity.

DssThreadId Identity of the calling unit. The identity should be unique at
the site from where the call is issued. The calling unit should assign all
its calls with the same identity if it is performing thread operations such
as locking.

MIOut_Interface The MM (??) holding the arguments to the abstract op-
eration. This is the data which the CA later will ask the EVM to marshal
(using MIOut_Interface methods).

The outcome, OpRetVal, passed as the return value of the the methods invo-
cation (see section ??), is one of the following types and instructs the mediating
layer what to do with the calling unit:

DSS_PROCEED The original operation should be performed by the thread
immediately.
**DSS_SKIP** The operation performed by the CA has executed the operation. The thread should skip the operation invoking the DSS and continue with the next instruction.

**DSS_RAISE** An error has occurred, the operation cannot be performed (commonly due to a perm fault).

**DSS_SUSPEND** The thread shall be suspended. The thread will later be resumed and will at that point either redo the operation or skip it.

A suspended calling thread will, on a successful operation, eventually be woken up again through one of the *resumeThread* methods of the EM interface (2.5.2, 2.5.3 or 2.5.4). It is of great importance for the correctness of the consistency protocol that the suspended unit is stored and woken up when instructed to.

### 2.1.4  *doAbstractOperation - relation*

```c
OprRetVal doAbstractOperation(CA_Interface* ca,
const AbsOp& aop,
const DssThreadId& id,
MMOut_Interface* pout)
```

The second method for performing an abstract operation is similar to the first with the difference of another CA as an argument. This method is used when a protocol operation involves two entities and performs a relation operation (typically an atomic binding of two entities). The parameter representation is the same as for the above operation, except the additional *CA_Interface*; The CA acting as an argument to the relation operator.

The behavior is exactly the same as for the single-case version of the method with the return codes having the same meaning.

### 2.1.5  *manipulateCN*

```c
MCNRetVal manipulateCN(const CNop& aop,
opaque& argument)
```

A coordination type is selectable when an abstract entity have been chosen. Depending on the kind of coordination a programmer may be able to perform different operations on the coordination set itself. For instance one might want to relocate the coordinator to another site or replicate the coordinator to several sites. Depending on the coordination type one or more operations are possible to perform using this interface method.

**CNop** The type of operation. Depends on the coordination type if it is possible to perform the operation.
opaque argument The argument may represent a remote process or a value. It is dependent of both the operation and the type of structure.

The MCNRetVal can be either MCN_OK if it might be possible to perform the operation, MCN_OP_FAIL if the DSS detects it is not possible to perform the selected operation or MCN_ARG_FAIL, if the argument or operand is wrong in any sense. Note that although MCN_OK has been indicated it is not assured that the operation will succeed, only that it is initiated successfully.

2.1.6 instrumentMM

| function | MCMMRetVal instrumentMM(const MMalg& alg, const MMop& op, opaque& argument) |

The instrumentMM offers a channel to the memory management module of a CA. The memory management layer is an implementation of the component based DGC schema. The idea is that memory management is handled by one or many DGC algorithms in conjunction to determine the total status of a reference. During the CA lifetime the set of algorithms can be altered as well as settings for a specific algorithm instance, hence not only can the memory management module be addressed but also a single algorithm.

The purpose of such instrumentation is to tune the memory management part to cope with failures or speed up the identification of the correct status.

MMalg The algorithm that is to be manipulated or removed.

MMop The operation, either a remove-the-algorithm or write of an algorithm-specific setting. The DSS will check the operations and only carry out valid once.

opaque Argument An integer value. The DSS will check if the value is valid and pass it on to the reference consistency module.

The value of MCMMRetVal can be either MM_OK or MM_FAILED. No other information is given so it is up to the Mediating Layer (of the EVM) to implement protections and, if wanted, failure messages.

2.1.7 getDssDGCStatus

| function | DSS_GC getDssDGCStatus(void) |

Garbage collection in the DSS uses the notion of roots for entities that should be protected from reclamation by the EVM. The memory management module, tracking references to an entity, of the CA is independent of the rest of the CA. However, the status is also depending on the consistency protocol, for instance the entity state may be residing on the process, thus removing it would be disastrous as perhaps the only complete entity structure would be lost. Memory management is initialized at CA creation and afterwards its interface for
getting the status is exposed from the CA. This method retrieves the current memory management information and returns it to the EVM. The status is also dependent on the coordination type chosen which may have impact on the performance as for more complex CAs is more complex to deduce the status. Note that the status of a CA can change if new references are imported. Therefore, it is crucial that the DSS is not activated (for example through an invocation of a heart beat) while a decision based on retrieved DGC status is taken.

The taxonomy for the garbage collection status of a CA is simple and efficient. DSS_GC, the returned status of a CA, can be of one out of the four following kinds, depending on both consistency protocol status and/or memory management status.

**DSS_GC_ROOT** The CA is a strong root for the EVM. This status means that the entity may not be removed, else the behavior for the entire entity and all the CAs is undefined. Note that the DSS might pick a CA for safe-harbor the entity structure, this CA is typically chosen where the coordinator is residing or has been residing. This results in the CA considering itself a root yet on a conceptual level it is not. This a pure implementational decision.

**DSS_GC_NONE** Nothing from the DSS prevents the EVM from removing the CA.

**DSS_GCLOCALIZE** The CA instructs the EVM that there exists no external references to the entity and it is safe to localize it into the VM and hence remove it from the DSS. Localizing means that the complete entity structure is locally available and thus it is safe to revert the entity into being regular accessed.

**DSS_GC_WEAK** The CA is not a strong root according to the memory management module of the CA but the state of the consistency protocol module prevents it from being a non-root. The property might though be changed using the clearWeakRoot method. Note that the protocol only guarantees to signal when it is assured the EVM cannot know if the CA is a weak root. Hence a protocol will assume that, for instance, a synchronous port-send is completed before root status is checked (should be guarded by a suspended thread). If the CA is checked during the wait for a reply it might not signal DSS_GC_WEAK as the EVM should be aware of this anyway. Note that the weakness property is subordinated the root property and the localizing property.

### 2.1.8 clearWeakRoot

```cpp
bool clearWeakRoot(void)
```

When a CA has informed the calling unit it is a weak root, see above, this method can be invoked to try to remove the obstacle preventing the CA from being removed. The returning boolean indicates if the function succeeded to
remove the weak state and changed it into being a non-root. If true the calling unit might safely remove the entire CA as it has become a non-root, else it must be treated as a regular strong root. The effect of this operation might though be seen later on.

2.1.9 dispose

void dispose(void)

The last of the garbage collection methods are actually the disposal of the of a CA representative. When method (2.1.7) returns DSS_GC_LOCALIZE or DSS_GC_NONE the only action needed is to dispose the CS which guarantees to remove all the traces of it in the DSS. Premature removal might of course lead to an undefined behavior for the Entity.

2.1.10 getFaultState

FaultState getFaultState(void)

The state (not the entity structure) of an entity is defined by two of the components building it. First, failures affecting the CLS can affect the functionality of the entity. A loss of the resolver(s) would make the entity useless for further usage; unless the CLS is designed to cope with such failure. Second, remote process failures that affects the consistency protocol can corrupt the entity. An example would be the single complete entity structure located at a certain site, if this certain site is lost due to a failure, the consistency protocol would fail to function properly. Each CA offers an interface for failure reporting. The method returns the current fault state of the CA (instance). An entity reports problems for its two sub components, the coordination type in the form of coordinator problems, and operation handling in the form of operational problems. The fault states are derived from remote process information, resulting in two possible values: Temp and Perm. This results in the following types of possible failures:

FS_NO_FAULT The entity functions correctly.

FS_PROT_STATE_TMP The consistency protocol experiences a fault that prevents it from execution. The fault is of a temporal nature and might go away later.

FS_AA_COORD_TMP_UNAVAIL The coordinator(s) of the CLS are temporary unavailable. This will most likely affect the functionality of the consistency protocol and thus result in a State-Temp fault as well.

FS_PROT_STATE_PRM_UNAVAIL The consistency protocol experiences a permanent fault caused by the permanent loss of the state. The entity will never function correctly again.
FS_CLSCOORD PRM_UNAVAIL The coordinators(s) are permanently gone. All functionality offered by them (the coordinators) are permanently gone. The Entity is most likely destroyed.

FS_CLSCOORD REMOVED The coordinator(s) has been removed implicitly by the EVM. Otherwise the same effect as previous.

2.1.11 setFaultState

```cpp
void setFaultState(const FaultState& fs)
```

When a failure is detected the DSS will inform each CA of the fault-state transition, this is done internally in the DSS and this method should never be invoked by the EVM. The DSS will check the current fault state and add the new information if necessary. If the EVM has registered interest in the information the coordination facility can tell the EVM when a new fault state has been entered. Thus, a CA implements both polling and pushing semantics for its fault-reporting functionality (see 2.5.5).

2.1.12 setRegisteredFS

```cpp
void setRegisteredFS(const FaultState& fs)
```

Since the failure mechanism of the DSS is two-fold, reporting to each involved CA and also their respective mediator, avoiding report flooding is of importance. To facilitate the behavior of failures the DSS will only notify the EVM on failures it has specifically registered interest for. Note that interest for multiple fault-states can be present at the same time. The combination can be determined by masking the the type of faults into a wanted fault-state interest.

2.1.13 getRegisteredFS

```cpp
FaultState getRegisteredFS(void)
```

The purpose of this method is simply to retrieve the current registered fault interest the MAP has interest in.

2.1.14 marshal

```cpp
void marshal(DssWriteBuffer* buf,
    const CAMarshalFlag& flag)
```

The serializing method of a CA is invoked when a distributed CA is sent to another site. There are two major tasks, the first is to gather enough information to build an entity instance (CA) on the other side, or if one already existed, silently merge the information into that one. The second task is to gather memory management information to enable memory management. The serialized data is put into the DssWriteBuffer which is assumed to have enough space.
The DSS supports three marshaling modes, depending on the initiator of the marshaling and also on the CLS of the entity. The three modes are triggered by the CA_MarshalFlag and represent:

**CAMF_ORDINARY** Ordinary serialization of the CA representation. This mode should only be used when the marshaling was initiated from the DSS. The data is written to the DssWriteBuffer buffer area. The call assumes that the needed amount of bytes is available in the passed buffer.

**CAMF_FREE** When marshaling is initiated from the EVM a fully marshaled representation of the CA is produced and written to the buffer. However, the representation does not contain any memory management information and the entity structure is thus made persistent. The call assumes that the needed amount of bytes are available in the passed buffer.

### 2.2 unmarshalCA

```cpp
bool unmarshalCA(CA_Interface&* CA,
                 DssReadBuffer* buf,
                 const CA_UnmarshalFlag& flag,
                 AE_Name& ae)
```

Deserializing of a CA structure is the inverse of the marshal method 2.1.14. The unmarshaling routine, however, is not a member method of the CA interface. The function is speculative; meaning that if an instance of that CA does not exist, one is created, and otherwise the transported data is silently merged into the existing structure. The boolean return code indicates to the EVM of whether the CA existed before or not, if none existed the EVM has to create a LR and connect the EM to the unmarshaled CA through method 2.1.1. The AE_Name returned (through call by reference semantics) tells the EVM of which abstract entity was chosen for the CA. A CA_Interface is always returned through call by reference semantics. As with the marshaling routine there are different modes of unmarshaling:

**CAUF_ORDINARY** Builds a CA representation from information found in the buffer. If a representation of the CA already exists in the DSS, the serialized information will be transparently merged to the already built one, otherwise a new one would be built. It is assumed that the passed buffer contains a fully serialized version of the CA.

**CAUF_FREE** Builds a CA representation from information found in the buffer. If a representation of the CA already exists in the DSS, the serialized information will be transparently merged to the already built one. This option indicates that no memory management information was stored in the buffer otherwise it should contain the same fully serialized version of the CA as in the case of CAUF_ORDINARY.
2.3 createCoordinationSet

CA_Interface *createCoordinationSet(const AE_Name& name,
 const ProtName& prot,
 const CLS_Name& cls,
 const int& GC_annot)

The creation of a shared entity is, for the most cases, independent of the
entity type. Typically the MAP has statically paired an entity type with a
predefined AE, an consistency protocols, coordination strategy and memory
management information. Whenever an entity is globalized a shared entity
should be created with these specifications, using this method, and an entry-
point, or reference, to that entity is returned in form of a CA_Interface object.
Using this interface the EVM can later perform operations on that specific
entity. The function returns the DSS handle to the entity instance, the CA.

Almost all parameters are enumerators representing the different compo-
nents needed to create a shared entity. The memory management parameter is
a bit-wise integer description of a number of algorithms.

AE_Name The enumerator representation of the chosen abstract entity (AE).

ProtName The representation of which consistency protocol to use. Must be
valid for the chosen AE.

CLS_Name The enumerator representing the chosen CLS.

GC_annot The bit-wise description of which distributed garbage collection
algorithm(s) (DGC) to use. Information on DGCs is found in section ??.

To complete the creation of the shared entity and the binding of the CA to
the specific LR, the EVM has to create a Entity Mediator for the entity and
inform the CA about this.

2.4 instrumentDssMM

MMRetVal instrumentDssMM(const MMalg& alg,
 const MMop& op,
 opaque& argument)

Instrumentation of the memory module in the DSS is achieved through this
function. Instrumenting an algorithm using this function affects the default
behavior, of memory management, for newly constructed CAs. This is can for
instance be the default set of algorithms if none chosen or the behavior of a
special algorithm. The parameters and return codes are the same as for 2.1.6.

Instrumentation of the memory module is used to achieve faster reclamation,
ability to cope with failures or less memory used by memory management part
of a CA.
2.5 Class EM_Interface

The entity mediator interface is defined in the api-header file and all methods must be implemented by the EVM. It is the available interface the CA has to perform abstract operations or resume operations on. The EM_Interface is the EVM part of the one-to-one mapped, together with the CA, entity representation on the local process and can be regarded the abstraction, or interface, to the LR. The Entity Mediator is for example responsible for securing the local representative during garbage collection, however, the interface only deals with operations explicitly performable by the CA, thus other issues are totaly EVM dependent. Included in the interface, as an attribute, is a pointer to the corresponding CA. This is just an implementation detail simplifying the interface.

2.5.1 doAbstractOperation

```c
MMOut_Interface* doAbstractOperation(const AbsOp& aop,
const DssThreadId& id,
MMIn_Interface* pstin)
```

The abstract operations callback implements access to the LR of an entity, for the CAs. The abstract entity chosen defines a set of abstract operations that it may perform. It is the doAbstractOperation that is responsible for implementing the EVM-specific part of the operation, i.e. the real operation.

A callback to the doAbstractOperation receives three parameters; out of these, one is used to classify the operation, the Abstract Operation. This, along with the chosen AE, defines what the CA wants the LR to perform. The other two parameters are the original calling thread and the argument for the EVM-specific operation. A MMOut_Interface (see 4.1) is returned to the DSS. Since the return value can be passed over the net it has to be encapsulated in a Message Mediator. The return of 0 (NULL) indicates that the callback did not produce any resulting value. The parameters are:

- **AbsOp** The abstract protocol operation. used to pinpoint the correct EVM operation.
- **DssThreadId** Identity of the calling thread.
- **MMIn** A pointer to the mediator holding the argument to the EVM operation.
- **MMOut** A pointer to the container for the returned data that is the result of the operation, which may later be marshaled if the consistency protocol wants it to.

2.5.2 resumeProceduralThread

```c
WakeRetVal resumeProceduralThread(const DssThreadId& id,
const ResumeCode& rc)
```
Operations on an entity can return values that instruct the calling thread to suspend itself. Suspensions is commonly a tool for the abstract entity to preserve a semantical property, i.e. guaranteeing casual order. Later, when the operation has been carried out, the entity instance needs to resume the suspended threads. It is of great importance that the application respects the return values from the operations and actually suspends the calling unit. Furthermore, it is of course a must to resume the threads when instructed to.

The call instructs the EVM to wake the thread identified by the passed caller identity (DssThreadId). The call is further defined by a Resume Code that describes how the thread should be resumed. If the operation on the entity would fail for any reason, suspended threads would be woken up with this call together with a special failure resume code. The resume codes are:

**TRC_PROCEED** The calling thread must redo the operation on the entity that suspended it.

**TRC_SKIP** The operation has been conducted by the DSS, the thread should continue with the next instruction.

**TRC_FAIL** The operation has failed and the thread should take appropriate counter-measurements.

The return value, `WakeRetVal`, of the method indicates to the consistency protocol (of the CA) what happened with the resume call:

**WRV_OK** Everything went fine.

**WRV_NO_THREAD** The thread could not be found by the EVM, the instructions was never executed.

**WRV_NO_OP** Nothing was performed by the method, the resume code indicating what was instructed to do was illegal, in some way.

### 2.5.3 `resumeFunctionalThread`

```c
WakeRetVal resumeFunctionalThread(const DssThreadId& id,
                                   MMin_Interface* mmin)
```

Abstract Operations can be defined to be functional. If a thread is suspended when performing a functional operation, it will later be resumed and handed a return value. The passed value is the value returned from the abstract-operation callback where the operation was actually executed.

The functional method implicitly informs that the operation was completed successfully from the perspective of the DSS. A return value in the form of a Message Mediator (pointer) is passed, containing the return value. The call instructs the EVM to wake the thread identified by the passed Thread identity (DssThreadId).

The return values are the same as for `resumeProceduralThread` (see 2.5.2).
2.5.4 resumeAtomicThread

\begin{verbatim}
WakeRetVal resumeAtomicThread(const DssThreadId& id)
\end{verbatim}

Resuming an atomic thread means that the original operation must be executed immediately; when the method call returns, the operation must have been executed.

The call instructs the EVM to wake the thread identified by the passed Thread identity (ThreadId).

The return values are the same as for resumeProceduralThread (see 2.5.2).

2.5.5 reportFaultState

\begin{verbatim}
void reportFaultState(const FaultState& fs)
\end{verbatim}

The last member of the EM methods is failure reporting. The entity instances, CAs, detects network disruption or failures that affects their ability to function correctly. If the functionality would be restricted in any way, this can be detected by the application. It is then up to the application to do correct counter actions, and try to isolate the failure so that it does not cause an entire application failure.

As mentioned in 2.1.12, to avoid flooding the EVM with reports of fault state transitions the interest for failures must be given to the corresponding CA. If a fault occurs that matches the given interest the CA will report back to the EM. Furthermore a match will erase the interest, and a new must be reinserted. Note that there can at most be one interest per CA, although a compound of many. If a new interest is installed on top of an old, the old will be completely overwritten.

3 Memory Buffers

Data transfer between processes must be serialized into a byte stream. It is the responsibility of the sender to produce a serialized format enabling the receiver to rebuild the data. Accordingly, passing Message Mediators between processes ends up in actually passing streams of bytes.

Abstractions of buffers are used to represent the start, and the end of byte streams. Whenever byte-stream data is passed to the EVM from the DSS or from the DSS to the EVM, it is encapsulated in a buffer Abstraction.

The Abstraction representing the source of a byte stream is called a DssWriteBuffer. The receiving end is called a DssReadBuffer. The source buffer exposes an interface for writing data into the stream, whereas the receive buffer has an interface enabling readings from the stream.
Figure 1: Time diagram showing displaying how the write buffer is passed back and forth between the DSS and the EVM.

3.1 Class DssWriteBuffer

The DssWriteBuffer represents an interface to a continuous, writable, data area. Buffer instances are used in interfaces routines passing rights to write data between the MAP and the DSS. The write buffer can be seen as a pipe to a recipient process. Data written to the buffer will appear at the other process in the order it was written.

Writing to the buffer can be done in two modes. First, bytes can be written one at a time to the buffer using (3.1.2). Second, the buffer offers an interface for copying blocks of bytes to the buffer(??). This is commonly used to overcome the interface overhead imposed by the virtual methods and when larger chunks of data can be directly copied into the buffer.

The write buffer can have space limitations. Method (3.1.1) returns the amount of space left in the buffer.

Figure 1 displays a typical usage case of a write buffer. At instance (1), the DSS asks the EVM to serialize a MM. The EVM starts serialize the data structure (2). A shared entity is found, and the DSS is asked to produce a serialized representation into the buffer(3). When done, the write buffer is handed back to the EVM, and serialization of the tree continues(4). At point (5) the MM is fully serialized and the write buffer is passed back to the DSS.

3.1.1 availableSpace

int availableSpace(void)

Returns the total amount of space, in bytes, left in the buffer.

3.1.2 putByte

bool putByte(unsigned char byte)

Inserts byte at the end of the buffer, after the previous write attempt to the buffer. If enough space was available in the buffer and byte was successfully written the call returns true, else false.
3.1.3 copyToBuffer

```c
bool copySafeToBuffer(unsigned char* ptr, 
                     size_t len)
```

The method performs a memory copy from a passed data area reference to its internal buffer representation. The source data area is represented as a pointer to the start of the memory block(ptr) and the size of the data area(len). The call returns the number of bytes successfully copied to the buffer, which is the smallest amount of the available space and the requested size.

3.2 Class DssReadBuffer

The DssReadBuffer implements an interface to a data area containing serialized data structures. The buffer interface is used whenever serialized data is passed between the DSS and the EVM.

Similar to the DssWriteBuffer the read version offers two “modes” for accessing its internals. Either through reading data, one byte at a time(3.2.3) or by copying chunks of data(3.2.4) out of the buffer.

Reading chunks of data out of the buffer is seen as speculatively from the buffer perspective. The buffer-internal notion of what data that is consumed or not does not change. The call 3.2.2 commits the read and changes the buffer state. Note that the single byte reading interface commits the read of each single byte. Thus, extra care must be taken if the two methods are used simultaneously.

The buffer can contain more than one unit of data, i.e more than one message. The method returning the amount of data in the buffer(3.2.1) returns the total amount of readable data, without taking in consideration the internal structure of the serialized information. Thus, it is of great importance, for the consistency of the unmarshaling routines, that readers from a buffer reads the data intentionally produced for them.

Furthermore, data is transported in frames. Where a frame contains information produced by both the DSS and the MAP. The interaction between the two layers is depicted in figure 1. Figure 2 show the possible layout of a frame. It can be seen how the data originating from the two layers are interleaved. Consequently, it is of great importance that each layer reads all data produced for it by the layer at the sending process.

3.2.1 availableData

```c
int availableData(void)
```

Returns the total amount of data which can be read out of the buffer. Since the buffer lacks any knowledge of the internal representation of the data, the returned value is the sum of all data units written into the buffer.
Figure 2: Internal layout of a message. Note how EVM data and DSS data is mixed in the block of bytes. The read buffer has no explicit knowledge of this structure. Consequently, it is the responsibility of the reader to respect the layout.

3.2.2 commitRead

```c
void commitReadSafe(size_t len)
```

The call commits a read to the buffer. Internally, the buffer records that `len` number of bytes has been read out of the buffer. I.e. the amount of available data is decreased by `len`. The methods only checks that `len` amounts of data could be read, not which layer should have read the data.

3.2.3 getByte

```c
const unsigned char getByte(void)
```

Returns the next byte to read from the buffer. This decreases the amount of available data by one (1). If no data is available an exception will be thrown, `EXCEPTION_NO_DATA`.

3.2.4 copySafeFromBuffer

```c
int copySafeFromBuffer(unsigned char* ptr,
            size_t     len)
```

The call copies `len` number of bytes from the buffer into the data area located at `ptr`. The call returns the amount of bytes actually copied. No more data will be copied than what is available in the buffer. Note that the call does not commit the read, this is done by the method(3.2.2).

4 Message Mediators

Arguments to an abstract CA operation or the result of an EM callback are referred to as Message Mediators. The representation of a MM is internal to the EVM and hidden from the DSS by a framework of containers, called Message Mediator interfaces. Whenever data is passed as argument to an abstract operation or returned as an result from a callback, it must be encapsulated
in a Message Mediator. All functionality of Message Mediators must thus be implemented by the EVM.

The Message Mediators are transported in buffers and handed over to the DSS during unmarshaling of a message. The Message Mediator represents something atomic, it is either fully received or not received at all.

The DSS tries to control its resource usage in general, and control its memory usage in particular. The buffers available for transporting data are limited. The size is configurable, and commonly set to a fairly large value. For this reason, it can happen that the serialized representation of a message does not fit into a buffer.

To cope with the problem of too large serialized representations of messages, the message mediator has to implement a strategy called incremental marshaling. Incremental marshaling is when a EVM node can be split up into a sequence of chunks. Each chunk of data is frame.

The Message Mediator can be seen as a pipe from one process to another, serializing a language graph. The two ends are referred to as Message Mediator Out, or MMOut, and the Message Mediator In, or MMIn. The MMOut communicates with a MMIn by passing chunks, or frames. From the EVM's perspective, the DSS opens a pipe between the out and the in mediator. The pipe guarantees in-order delivery. It is possible that the pipe is lost. In such a case the MMOut is asked to redo the whole transfer. Note that when an operation can be performed locally, the MMOut is asked to return a MMIn representation of itself, thus no unnecessary serialization is performed. In rare cases the opposite is also possible.

4.1 Class MMOOut_Interface

The starting point of an EVM argument transfer is represented by a MMOut. The MMOut is asked to marshal itself using the method(4.1.1) until either no buffer space is left or the entire EVM argument has been marshaled.

The MMOut is responsible for marshaling its contents and if problems should be encountered, be prepared to marshal its contents again. The EVM argument it represents must thus be valid for marshaling until the DSS disposes the MMOut.

The MMOut is allocated by the EVM as a container of a message and a reference is passed to the DSS. Still, the full responsibility for the object is left at the EVM. Through one, or more calls to 4.1.1 the contents of the MMOut is transferred. If all data are correctly received at the receiving side the MMOut is asked to dispose itself (4.1.3). However, if the transfer is not successful, the MMOut will be asked to reset (4.1.2) itself and then be marshaled again.

The lifetime of a MMOut is depicted in figure 4.1. From the point where the MMOut is passed to the DSS, and until it fully received at the target site, the data must be kept alive.

4.1.1 marshal
Figure 3: Life time of a MMOOut. The MMOOut is created and passed as an argument to the abstract operation (1). The DSS waits for a possibility to write the message on the channel connected to the destination site. When the DSS can write data on the channel, the MMOOut is asked to marshal itself (2) into a buffer. When the marshaling is done (3), the message is transferred to the remote process (4). When the message is acknowledged (5), it is known that it is fully received; at this point the MMOOut is asked to dispose itself. This ends the lifetime of the MMOOut, from the perspective of the DSS.

```cpp
bool marshal(DssWriteBuffer* buf)
```

The call asks the object to marshal the data it contains into the write buffer `buf`. The marshaled data will be treated as a serialized chunk by the DSS and atomically delivered to the MMIn on the receiving side.

If the buffer is unable to receive the full serialized representation, i.e. the message has to be sent as a sequence of chunks the call should return false. This indicates that the DSS must perform further calls to the object for transfer the full representation of the message contents. Otherwise, if the method should return true.

### 4.1.2 resetMarshaling

```cpp
void resetMarshaling(void)
```

This method will be called by the DSS if the message is to be resent. The object must be ready to marshal the data again. A MMOOut in the middle of an interrupted marshaling can be asked to reset itself.

### 4.1.3 dispose

```cpp
void dispose(void)
```

The call tells the object that the data is fully transferred and the MMOOut is no longer needed.
4.1.4 loopBack2In

```c
MMIn_Interface* loopBack2In(void)
```

The function should produce a new MMIn from the MMOut. Preferably, this is done without marshaling the data. This functionality is used when messages are routed internally by the DSS, typically when a protocol realizes that the data should be delivered at the current site. When this method has been invoked the EVM is hinted that the specific MMOut is not going to be used anymore and can expect a `dispose` invocation.

4.2 Class MMIn_Interface

A Message Mediator In, `MMIn`, represents the incoming end point of a message transfer. At some site there exists a start point for the transfer, represented by a MMOut. The MMOut produces serialized chunks given to the MMIn. From these frames the MMIn rebuilds the message.

The MMIn is the representative for a received message from the perspective of the DSS. First as the builder of the message, and then as the container for the message. When the message is fully received (signaled to the DSS through the method 4.2.1) it is passed to its target. When passed like this it is representing the fully built message.

Note that since a message can be received in multiple chunks, a semi-built message must survive possible garbage collections.

A MMIn is first allocated by the DSS using the function 4.3. Now it acts as the endpoint for a message transfer and receives one or more serialized chunks. When fully received it is passed to its target Entity Mediator and after this disposed. It can be disposed before being passed to the Mediator if a problem has been experienced by the DSS, thus a new MMIn will be created when the transfer is restarted.

4.2.1 unmarshal

```c
bool unmarshal(DssReadBuffer* buf)
```

The call tells the object to deserialize the data found in the `buf` and store the result internally. A complete serialized chunk is available in the buffer. The call should return whether the message is complete (true), or if more chunks are to be expected (false).

4.2.2 dispose

```c
void dispose(void)
```

The call tells the MMIn it is no longer needed by the DSS and that it can safely be disposed.
4.2.3 loopBack2Out

MMOut_Interface* loopBack2Out(void)

The function should produce a new MMOut from the MMIn, containing the same data. Preferably, this is done without marshaling the data. This functionality is used when messages are routed by the DSS, typically when forwarding a message from the site to another as a consequence of a protocol operation. When this method has been invoked the EVM is hinted that the specific MMIn is not going to be used anymore and can expect a dispose invocation.

4.3 createMMInContainer

MMIn_Interface* createMMInContainer(void)

The DSS has received a new message and needs a MMIn for building the VM dependant representation. The function should return a fresh MMIn, which later will be asked to unmarshal serialized data. This is implemented by the EVM.

5 The I/O Service

The DSS needs to transport blocks of bytes between processes. Throughout this document, channels have been the abstract description of a medium enabling transfers of bytes between processes. How a channel is implemented is not of interest to the DSS. It only has to be able to transport blocks of data from one process to another process in FIFO order.

Channels are commonly asynchronous and have a limited throughput. implying that they will accept a certain amount of bytes, and then be blocked until the data has been delivered to the recipient. A socket is a typical channel. Furthermore, the interface to a channel is typically operating system dependent.

Commonly the EVM already have functionality for handling I/O over channels, the DSS is designed to use this kind of service. The DSS comes with an abstract notion of a channel; data can be read and written to a channel; a channel can tell when there is data to either read or write to it. It is the responsibility of the EVM to provide the I/O handling for the DSS, exposed in the form of channel objects.

5.1 Class CallbackHandler

CallbackHandler objects are used as interfaces between the I/O handlers of the EVM and the transport mechanisms of the DSS. The CallbackHandlers are implemented by the DSS. The I/O handlers, described later, communicate with the DSS through callbacks; when an I/O handler needs to signal to the DSS, it
invokes the callback method of a CallbackHandler. The DSS guarantees it will only access the channels after it has been notified through a callback.

When the DSS register itself for callback, instances of the CallbackHandler class is passed as argument. As long as the interest for the callback lasts, the EVM has to store a reference to the CallbackHandler. However, creation and deletion of CallbackHandlers is the responsibility of the DSS.

5.1.1 invoke

bool invoke(void)

This method should be called to signal to the DSS that an event has happened.

5.2 Class DssTransportChannel

The DssTransportChannel class is used as an encapsulation of a transport channel resource, like a TCP socket. All channels passed from the EVM layer to the DSS must be encapsulated in DssTransportChannel objects. Instances will internally in the DSS be used to send messages to and receive messages from the remote site where to the channel is connected. The DssTransportChannel is implemented by the EVM and used by the DSS.

The state of the channel, i.e if it is lost, should be returned from the read and write methods using ErrorClass enumerators:

EC_OK The operation was successful.

EC_GO_AHEAD The operation failed, but will most likely be successful if redone immediately. On Unix system this is the socket error EINTR.

EC_CONTINUE_LATER The operation failed, the channel can repair itself later.

EC_LOST The channel is lost, it will not repair itself.

5.2.1 readData

unsigned int readData(unsigned char* buf,
                     size_t len,
                     ErrorClass& err)

The method is used to copy len bytes from the channel to the memory area located at buf. The ErrorClass err indicates if any problem has been experienced during the read operation. The method returns the actual amount of data copied from the channel to the buffer.
5.2.2 writeData

\[
\text{unsigned int writeData(unsigned char* buf,} \\
\text{size_t len,} \\
\text{ErrorClass& err)}
\]

This method is used to write \text{len} bytes from the buffer found at \text{ptr} to the stream. The amount of data actually written is returned together with the status code \text{err}.

5.2.3 closeChannel

\[
\text{void closeChannel(void)}
\]

Used by the DSS when the channel is no longer needed. The DSS will not access the channel after it has been closed. Any registered read or write handlers must be automatically deregistered.

5.2.4 registerWriteHandler

\[
\text{void registerWriteHandler(CallbackHandler* handler)}
\]

The DSS register for a write interest. Whenever there is space enough to write data to the channel the WriteHandler \text{handler} should be invoked. It is assumed that the Channel only holds one write callback handler, it should thus not be allowed to perform multiple registrations without deregistering in between.

5.2.5 registerReadHandler

\[
\text{void registerReadHandler(CallbackHandler* handler)}
\]

The DSS register for a read interest. Whenever there is data to read from the channel the ReadHandler object \text{handler} should be invoked. It is assumed that the Channel object only holds one read callback handler, it should thus not be allowed to perform multiple registrations without deregistering in between.

5.2.6 unregisterRead

\[
\text{void unregisterRead(void)}
\]

Unregisters a read interest from the channel. The ReadHandler object must not be deleted. This is the responsibility of the DSS.

5.2.7 unregisterWrite

\[
\text{void unregisterWrite(void)}
\]

Unregisters a write interest from the channel. The WriteHandler object must not be deleted. This is the responsibility of the DSS.
6 Representing Remote Processes

A known process the DSS can communicate with is called a remote process, see section???. Internally, the DSS builds a representation of the process, called a Remote Site or Site for short. Remote Sites represents the knowledge and the logical connection to a process. When data has to be transferred to the remote process, the Remote Site establishes a physical connection. The physical connection is then used to transfer the data.

The Remote Site representation is divided in two parts. First, naming or identifying and message sending. This is internal to the DSS and not exposed. Second, connection establishment, including addressing of the process. This is implemented by the EVM to enable application specific solutions. The two tasks are represented by two objects knowing of each other. The EVM implements a Site_Address, whose primary task is to establish connections to the remote process. The DSS exposes a Site_Name, used for interfacing to the internal representation of the remote site.

6.1 Transferring Site Representations

The DSS uses the Sites to identify entities and to represent locations. This information is passed between processes and must thus be serialized. To construct an instance of a Site representation, both the DSS representation and the EVM representation of the Site must be built, thus both the Site_Name and the Site_Address must be marshaled. The receiving side must rebuild the both representations.

It is not always the case that a site representation should be constructed when received. The DSS keeps at most one instance of a Site. If multiple copies of a Site are received, their information is merged into the already existing Site representation. Thus, the EVM can be told to conform to this schema and either construct a Site_Address from a serialized representation, or to dispose a serialized representation of an already existing Site_Address.

6.2 Connection Establishment

Creation and deletion of transportation channels is divided in two parts. First, when to do it; and second, how to do it. The DSS is responsible for the first task, to decide when to establish and close a channel. The second task, to decide how to bring up, and to bring down a channel is a responsibility of the EVM.

Dividing the responsibility like this gives the DSS the control over its allocated resources. When it needs to communicate, a channel will be established for it. Later, when the channel is no longer needed, the channel is closed. The DSS controls how many resources it has allocated. Finding the remote process, establishing a connection and executing any handshake protocol is the responsibility of the EVM. This gives the application the freedom to implement its strategies for finding remote processes, and to establish the connections. This
is preferable since the task of establish connections is inherently application
dependent.
There exists two types of connection sequences. First, the process has initi-
ated the connection to a remote process. This is called an Active Connection.
Second, the process is the recipient of a connection attempt. Referred to as a
Passive Connection. The different connections are described in the following
subsections. The issue of simultaneously connection attempts is discussed at
the end of this section.

6.2.1 Grants and Different Types of Channels

The DSS is designed to be able to communicate using arbitrary communication
medium. Which of the available mediums to choose is up to the connection
establish routines. This gives applications the freedom to choose a medium
suiting the network environment they are situated in. For example, to process
located at different LANS might prefer a TCP connection; while two processes
situated at the same process might prefer communicating using shared memory.
A set of possible transport mediums are exposed to the connection routines
to choose from. Commonly, a restricted number of instances of a transport
medium can be allocated simultaneously. For TCP channels, sockets is the
limiting factor, while for shared memory pages, the number of memory pages
restricts the channel resource. For this reason, the connection routines must
allocate the right to use a resource. This right is called a Grant, and represents
the preallocated channel.

Channels allocated for the DSS must be accounted for by the DSS. As men-
tioned, the DSS exposes a schema using Grants representing the right to use a
resource. It is important to properly allocate, and deallocate grants when using
channels on the behalf of the DSS. If not, the resource scheduling schema that
the DSS implements will not work properly. A successfully established channel
must have a grant associated with it. When the channel is passed to the DSS,
the grant must be passed, as a certificate of the right to use this channel. By
passing the grant to the DSS, the DSS takes over the control of the resource.
Allocated grants not needed, i.e. not handed to the DSS when the connection
is established, must be deallocated. Otherwise, the process will experience a
resource leakage, and eventually not be able to establish any connections.
Grants, discussed later in this section, refers to the granting of allocating
a channel. The DSS manages resources and channels are expensive and grants
are used to control the number of available resources.

6.2.2 Active Connection

When the DSS needs to construct a valid physical channel to a remote process,
it will initiate a connection attempt. This is referred to as an active connection.
The EVM will be asked to establish a channel to the remote process. It is then
up to the EVM to execute its own protocols for finding the remote process; to
decide upon which transportation medium to use; and to establish a channel of
Figure 4: A diagram showing the interaction between the DSS and the EVM during an active connection attempt. First (1), the DSS asks the EVM to establish a connection to the destination. Second, the connection establishment mechanism has to allocate a grant for the channel it will use (2)-(3). Third, the EVM executes its own connection establishment protocol (4) to the remote process. If a connection is established and the handshake is successful, the EVM hands the established channel to the DSS (5). The whole transaction is guarded by a timer (6), if the connection has not been successfully established before the timeout expires, it will be aborted.

If this succeeds, the channel is handed back to the DSS, and will then be used internally to transport data between the two processes. The interaction is depicted in figure 4.

The whole connection establishment is controlled by the DSS, using timers. More precisely, an active connection attempt is associated to the Site representing the remote process. If no progress is achieved within a certain timeout, the transaction is aborted. An aborted connection attempt will be retried later, it is assumed that temporary network problems is a possible cause to a failed connection attempt.

If the code trying to establish an active connection experiences a problem, before the DSS tells to abort, this should be reported back to the DSS. The Site initiating the connection can be told that the connection simply failed, this has the same meaning as an abort from the DSS. However, the Site can also be told that the remote process has crashed. This affects the connection establishment in that no retries will be executed.

6.2.3 Passive Connection

When a remote process wants to establish a connection to a DSS process, this is referred to as a passive connection. The identity of the remote site is not known during the establishment of the connection. Thus the connection attempt is not associated with a Site. Receiving, and accepting the connection is the responsibility of the EVM. When the connection is established is it handed over to the
Figure 5: A diagram showing the interaction between the DSS and the EVM during a passive connection attempt. The remote process initiates the connection and the EVM receives it (1). To be able to accept the connection, a grant has to be allocated (2)-(3). If the channel is properly allocated, and eventual handshake is successful, the channel is handed over to the DSS (4) as an asynchronous channel. Internally, the DSS will, by using its own handshake protocol, learn the identity of the remote process.

DSS, as an anonymous connection. Internally, the DSS will communicate with the remote process and make sure it is a proper DSS process. If no Site representation exists for the remote process, one will be constructed. Otherwise, the established channel is associated with the already existing Site. The sequence of a passive connection is depicted in figure 5.

Passive connections is different from active connections in that they are not known to the DSS until the channel is fully established and handed over. Thus, it is the responsibility of the EVM to abort connections not making any progress. Furthermore, failed passive connections should not be reported to the DSS.

6.2.4 Simultaneous Connections

A site in the middle of an active-connection attempt to a site S1 can simultaneously be the subject of a passive connection attempt originating from the same site S1. Thus two connection attempts can occur simultaneously between two sites.

The identity of the initiator of a connection is not known at the receiving side. Hence, detecting simultaneously connection attempts is complicated. First when the two processes has revealed their identities can eventual parallel connection attempts be detected. Since the true identity of the process on the other end of a connection is anyway learned when the allocated channel is handed to the DSS, the task of resolving simultaneously connections is the responsibility of the DSS. The connection establishment code should only focus on establish connections.
6.2.5 Sites as Resources

Internally in the DSS, sites are referred in a great extent. The number of references to a single site object is related to the number of shared entities. Commonly, the number of shared entities can grow extremely large, upholding a reference count to sites in the DSS is close to impossible. For this reason, the DSS manages its sites through a mark and sweep mechanism. Periodically, the DSS should be asked to check, through method 6.7, its sites to see which sites can be removed, and which to keep.

6.3 Class Site_Name

The internal representation of a remote process, called a site, exposes an interface through the Site_Name class. It acts as the identifier for a given site, and offers interfaces for communicating with the site object. The site offers three basic blocks of functionality. First, connection establishment, established connections are handed to a Site_Name. Second, registering for interest in faults. When the internal site changes its faults state, it can be instructed to call its EVM counterpart. Last, interface for actively changing the fault state of a site.

6.3.1 channelEstablished

```c
void channelEstablished(ConnectionGrant* grant,
                          DssTransportChannel* channel)
```

A connection request, EVM handshake, to the SiteAddress is successful. The call passes a resource right, grant and an encapsulation of the established connection, channel for the DSS to use.

6.3.2 connectionFailed

```c
void connectionFailed(void)
```

The call tells the site when a requested connection attempt has failed. The site will try to reconnect later, since the failure is regarded as a result of a transient perturbation. If the reason for the terminated connection attempt was a crash-failure of the remote site, the method(6.3.3) should be used.

The connection attempt is guarded by timers from the DSS, if no progress has been achieved within a time limit the connection will be terminated. However, it is good practice to inform the initiating site representation that the connection has failed. The timers guarding the connection attempt can be removed and no abort callback has to be initiated.

6.3.3 siteFault

```c
void siteFault(const SiteFaultType& fault)
```
This call tells the Site that the EVM has learned of a fault of the Remote Site. Internally, the DSS will detect partitioning faults, in the form of Temp Fail. However, since different types of crash-failure are application dependent, this is the responsibility of the MAP. Note that setting the state of a site to crash-fault will affect all entities originating at the Remote Site. SiteFailType has the following values:

**SFTGLOBALPERM** The remote site has crashed and will never be contactable again. The perm property is propagated, automatically by the DSS, to other remote sites that will change their perception of the crashed site. This state will thus eventually change the perception of all interconnected sites.

**SFTLOCALPERM** Similar to global perm in that the remote site will never be contactable again. However, this information will not be spread to other remote sites by the DSS. It is private to the local site.

### 6.3.4 registerFaultInterest

```c
void registerFaultInterest(const SiteFaultInterest& intr)
```

Sites offers a callback service for state change updates. A vector containing the state changes of interest are passed to the site. When a state change happens that the EVM has registered interest in, the method faultCallback (6.4.4) in the SiteAddress object is called. Registering for an interests overwrites previously registered interests. It is possible to register for a combination of the following state changes (A state change is defined as when the Site enters a new state):

**Connected** A physical connection has been established to the remote site.

**Disconnected** The physical connection to the remote site is lost.

**Temp Fail** A partitioning problem has been encountered.

**Perm Fail** The remote site has been defined as crash-failure.

**Normal** The site is normal again.

### 6.3.5 getFaultState

```c
FaultState getFaultState(void)
```

Returns the current FaultState of the Site.

### 6.4 Class SiteAddress

A SiteAddress acts as the interface to the site representation in the EVM for a DSS site. Its main responsibility is to establish connections to the process it represents. SiteAddress must be implemented by the EVM.
6.4.1 establishConnection

void establishConnection(void)

This method is invoked when the DSS wants to establish a physical channel to the remote process. When the physical connections established, it is passed to the Site_Name associated with the Site_Address.

6.4.2 abortConnection

void abortConnection(void)

The Site_Name asks for termination of an ongoing connection attempt. Connection attempts should be terminated as fast as possible. When the call returns the connection is terminated from the perspective of the Site_Name.

6.4.3 grantAquired

void grantAquired(ConnectionGrant* grant)

A grant has been allocated for the Site_Name. The grant was asked for by the getConGrantAsync function (6.10); when asked resources where scarce and the Site_Address had to wait until a resource was freed.

6.4.4 faultCallback

void faultCallback(const SiteFaultState& change)

This method is called when a state change of the DSS site representation matches the registered interest. The registered interest has in no way expired, but is still valid. The value change tells the new state.

6.4.5 dispose

bool dispose(void)

The DSS no longer needs the site and indirectly asks the EVM to dispose the address. The Site_Address has the executive right to decide if will be removed or not. If the call returns true, the site will be removed, false tells the Site_Name to not remove itself.

6.4.6 marshal

void marshal(DssWriteBuffer* buf)

A representation of the site is transfered over the network and needs to be marshaled. The Site_Address is asked to marshal a representation of itself into buf. This marshaled data will be used to build a Site_Address object at the receiving side using the Site_Address unmarshaling functions (6.5, 6.6).
6.4.7  stringrep

cchar* stringrep(void)

Used for debugging and logging purposes by the DSS. The DSS will concatenate the information from the Site_Address and the Site_Name to produce a more verbose representation. Should return an allocated string containing the textual representation.

6.5  unmarshalSiteRepresentative

Site_Address *unmarshalSiteRepresentative(DssReadBuffer* buf,
                                          Site_Name* name)

The function builds a Site_Address from its marshaled representation in buf and returns it. The object is the EVM layer representative for the supplied Site_Name name. Note that this function will only be called when no Site_Address representation exists for a Site_Name object. The function is declared as a function pointer which must be bound to an implementation by the EVM implementor.

6.6  disposeSiteRepresentative

void disposeSiteRepresentative(DssReadBuffer* buf,
                                Site_Address* addr)

A marshaled Site_Address representation of an already existing site is received. The marshaled data must be read out of the buffer buf to not corrupt further unmarshaling. Depending on the EVM implementation, more recent information regarding the address of the site may be found in the buffer. A reference to the already existing Site_Address object is passed in addr. The function is, as the previous, declared as a function pointer which must be bound to an implementation by the EVM implementor.

6.7  gcDssResources

void gcDssResources(void)

Representation of remote sites are allocated when needed and are internally seen as an ubiquitous resource. Deallocation of these objects is done by sweeping the whole DSS and recording the sites being needed. This can be a costly operation in terms of execution time; especially if the DSS holds many references to remote entities within the CAs.

The procedure will atomically scan all possible references to the internal sites, and deallocate the sites no longer needed. This will simultaneously close unwanted physical connections. The sites scheduled for removal will ask their EVM representatives, the Site_Address objects for removal using their dispose (6.4.5) method.
6.8 Class ConnectionGrant

The DSS is allocated a limited number of physical channels and guarantees it will never use more channels than allocated. To be able to communicate with arbitrary number of remote sites, a caching mechanism is implemented over the resources. If more channels are needed than what operating system has available, the DSS will time share the connections between the Sites. This class is implemented by the DSS.

A connection establishment, that is executed at EVM level, must therefore allocate a resource for each connection. Through a mechanism of grants, the right to use a resource is reflected from the DSS to the EVM layer. Each grant is represented as an instance of the ConnectionGrant class. Two functions are available for acquiring grants (6.9, 6.10). After acquiring a grant, the right to use the resource belongs to the EVM layer. To pass this right, the grant is passed as an argument to the channel established routines(6.11, 6.3.1). After passing the grant to the DSS, the DSS is now the owner of the grant, and not the EVM layer. Allocated grants not handed to the DSS must be deallocated using the freeGrant routine(6.12), to avoid resource leaks.

6.8.1 getConnectionType

ConnectionType getConnectionType(void)

Returns the connection type that the grant represents.

6.9 getConGrantSynch

ConGrantReturn getConGrantSynch(const ConnectionType type,
                                  ConnectionGrant*& grant)

Tries to acquire the right to use a resource of the ConnectionType type. If successful, the ConnectionGrant pointer grant is bound to the allocated grant. The function returns one of the following, that tells the outcome of the operation. Implemented by the DSS.

CGR_ALLOCATED A grant is allocated, the ConnectionGrant grant is bound to the allocated grant.

CGR_FAILED No grant was currently available.

CGR_ERROR The DSS does not have the type of connection installed.

6.10 getConGrantAsynch

ConGrantReturn getConGrantAsynch(const ConnectionType type,
                                  ConnectionGrant*& grant,
                                  Site_Name* name)
This function can be used by an active connection establishment attempt, if it can wait for resources. In difference from the synchronous allocation of grants(6.9), this function will put the requester in a queue if no resource is available. Eventually, when a resource is available, the caller(i.e the Site_Address) will be informed. Implemented by the DSS.

The Site_Name pointer name defines the site needing the resource. It will be used to find the corresponding Site_Address object.

CGR_ALLOCATED  A grant is allocated, the ConnectionGrant grant is bound to the allocated grant.

CGR_SUSPENDED  No grant was currently available. The request is put on hold until a resource is available. When available, the Site_Address object associated with the passed Site_Name name will be informed through the grantAcquired method(6.4.3).

CGR_ERROR  The DSS does not have the type of connection installed.

6.11 anonymousChannelEstablished

void anonymousChannelEstablished(ConnectionGrant* grant, 
                                 DSS_Transport_Channel* channel)

Successful passive connection attempts are handed to the DSS using this interface. The DSS_Transport_Channel argument channel represents the established channel to the remote site. Whereas the ConnectionGrant grant represents the right to use the resource. Note that the DSS takes over the right to use the grant, and that the grant should not be freed. Implemented by the DSS.

6.12 freeGrant

void freeGrant(ConnectionGrant* grant)

Deallocation an allocated grant. The resource is passed back to the pool of resources. Only grants that are not passed to the DSS should be freed. Implemented by the DSS.

7 Auxiliary

The first part of this document have described the objects and interfaces involved in distribution of entities. However, the DSS is also a middleware implementing its own messaging service. To make all this functionality to work the DSS needs to be carefully initialized. Furthermore a set of services and interface must be provided by the EVM. These have been collected in this section of auxiliary functionality.
7.1 initDSS

Site_Name* initDSS(const unsigned int& id1,
           const unsigned int& id2,
           Site_Address* sa)

The DSS needs to have an id and an address for identification of itself on
the network and also addressing itself. The user of the DSS is also responsible
for submitting two integers which will become its unique 64-bit id (see 6.3) when
initializing the DSS. The initialization procedure sets up all internal memory
structures, defaults and resources. Hence using the DSS without calling this
function is impossible. The function returns the Site_Name to be used when
communicating with other sites.

id1 First 32 bits of the unique id.

id2 Second 32 bits of the unique id.

7.2 heartBeat

void heartBeat(const int& TimePassedInMs)

Apart from establishing connections, sending and receiving messages the
dss needs timers to function correctly. Internally the DSS has a range of
different timer mechanisms for different tasks. To get these activates running
the DSS needs a notion of time and periodical invocations. Since the DSS
is a passive component is has no notion of time what-so-ever. It is therefore
the responsibility for the EVM to give the DSS the idea of time. Through a
heartbeat interface the EVM must invoke the DSS periodically. Furthermore
it must also tell the DSS of how much time has passed (in ms) since the last
heartbeat. Preferably, the DSS should be invoked as often as possible; this is of
course not possible so as a rule so the thumb, every 10ms, is a suitable interval.
If the heartbeat is invoked less frequent, performance will most likely degrade.
However, it will still function.

The TimePassedInMs is the amount of time passed since the last heartBeat
invocation in milliseconds. The first call to heartbeat should have zero ms as
time passed.

7.3 operateIntParam

ParamRetVal operateIntParam(const DSS_AREA& area,
                           const DSS_AREA_ID& id,
                           const int& param,
                           const int& arg)

The General API enables instrumenting the behavior of the DSS. This is on
the level of writing and reading default values for static components and current

38
values for dynamic components. These settings will affect the whole DSS and all structures using them. Parameters in general are accessed through a schema with Areas and Ids.

The different parameter values used by the static components of the DSS can be read and set. The parameter id is identified using an abstract Id schema, where each id is assigned a fixed value. A static parameter, in contrary to a dynamic parameter, is used for parts of the DSS which are live during the whole lifetime of the DSS, such as timeouts and default buffer sizes. Some parts of the DSS are only temporary available during runtime and settings for these parts are called dynamic parameters. A dynamic is created during runtime and it is referred to through a schema similar to that for static parameters. A unique DSS\_AREA pinpoints the area (static, such as hashtables) and the DSS\_AREA\_ID the specific parameter. For static parameters the DSS\_AREA identifiers is a special DSS\_AREA\_STATIC flag. The return codes for the operations are:

**PRV\_OK** Command was executed successfully.

**PRV\_TYPE\_ERROR** A problem with the value argument was encountered. Covers both arguments with wrong type and "out-of-range" problems.

**PRV\_AREA\_NOT\_FOUND** Area argument was incorrect.

**PRV\_DYN\_PARAM\_NOT\_FOUND** A dynamic parameter was not found.

**PRV\_STAT\_PARAM\_NOT\_FOUND** A static parameter was not found.

This function offers another kind of feature which may be triggered during compilation of the DSS. By enabling the dssLog feature (explained in the DSS source code tree) the DSS offers continuous logging of the ongoing execution inside the DSS. The logging utility can be instrumented to report different levels of events. More on how to make use of the dssLog is explained in the source root. Note that the logging utility is costly in terms of performance and should only be used during debugging.

### 7.4 operateStrParam

```c
ParamRetVal operateStrParam(const DSS\_AREA& area,
const DSS\_AREA\_ID& id,
const int& param,
const char* const str)
```

The second parameter function works exactly as operateIntParam 7.3 with the difference that the actual parameter param is a char-pointer.

### 7.5 GL\_error

```c
void GL\_error(const char* format,
...
```
GL_error is defined in the api-header file and must be implemented by the EVM. When the DSS encounters a EVM-caused fatal error it will call this function with a message string and variable number of arguments. This instructs the EVM that the DSS has stopped executing and the EVM should print the message and shut down. The function is declared as a function pointer which must be bound to an implementation by the EVM implementor. If no implementation is supplied, when linking the DSS to a EVM, the DSS will use a stub procedure which will shut the DSS down and print an error message.

7.6 GL_warning

```c
void GL_warning(const char* format, ...
```

GL_warning is defined in the api-header file and must be implemented by the EVM. It is a way of notifying the EVM (and programmer) that something unusual has occurred, although not fatal. The function is, similar to the above (7.5), accepting a message string and variable number of arguments. The severity of this “error” is not of the same magnitude as 7.5 so there is no need to shut the system down. The function is declared as a function pointer which must be bound to an implementation by the EVM implementor.