Making the Distribution Subsystem Secure

Zacharias El Banna, Erik Klintskog and Per Brand

June 29, 2005

Swedish Institute of Computer Science, Kista, Sweden

SICS Technical Report T2004:14
ISSN 1100-3154
ISRN:SICS-T–2004/14-SE

Contents

1 Introduction

This report describes the results in securing the Distribution SubSystem (DSS)[16, 6, 8] middleware library. The DSS provides generic distribution support for shared data structures/language entities on the level of first class references. Security for the DSS, is thus on the level of first class references. Internally this means unforgeable references, encrypted channels and secured consistency protocols as well as a robust implementation that is able to withstand an attack.

We have ensured that the DSS provides a reference secure shared data model. The DSS can be coupled to a centralized programming system to make a distribution-extended programming system[2], and with the security provisions this makes the distributed programming system reference secure. Alternatively the DSS can be used directly by the application level to provide a reference secure data-sharing model. Only the processes that have legitimately received a reference to a data structure can access the data structure. In addition, a process will only accept connections from processes that have legitimate reference to a data structure that is shared between the two processes.

1.1 Structure of the Report

The report is structured as follows. First, the rational behind and the implementation of the DSS are briefly discussed. This with the primary intention to put the reader in the context of the DSS (for further information about the DSS see [16, 6, 8]). Second, we describe the division of labor in respect of security between the DSS and an application/programming system that uses the DSS. This is the first contribution of this report. Third, we present the three major types of security threats a distributed application is faced with on the level of the DSS. These are later used in the report to validate the completeness of our solution. This is the second contribution of the report. Fourth, the internals of the DSS, and the extensions required to make the DSS secure (where practical achievable) is described. Fifth, the design of a secure DSS implementation is described. This is the third contribution of this work. The report is then concluded with an evaluation section, a section that arguments for a capability based model over an access list model, and a future work section. The contributions are summarized here:

- Model of division of labor in respect to security between the DSS and a programming system.
- Model of possible threat scenarios for a distributed computation.
- Design and implementation of a secure DSS middleware library that partly copes with the threat scenarios.

The future works section discuses further extensions to the DSS in order to handle(where practically achievable) all the thread scenarios.

2 The Distribution Subsystem - DSS

The Distribution Subsystem (DSS) is a language independent middleware library[1] for efficient distribution of data structures. The middleware library offers a novel language independent interface for sharing data structures. The interface is based on a notion of abstract entities and clearly separates functional concepts from non-functional concepts. The internal protocols that realize shared data structures, called *entity consistency protocols* are divided in three separate subcomponents. This approach to software design for entity consistency simplifies further protocol development and also allows for runtime protocol composition[16].

The DSS is not primarily intended to be used directly by application developers. Instead the DSS can be coupled to a programming system in order to realize a powerful distributed programming system. Central in the model is a clear division of labor between the (centralized) programming system and the DSS. Clearly the centralized programming system needs to be extended to be able to interact with the DSS and fulfill its part of the contract with the DSS. The programming system must now be able to marshal (or serialize) data structures and, in general, code. The work has been evaluated and results to date look good, both in terms of ease of coupling and performance [16, 6].

The DSS model, design and implementation, without security instrumentation is described more fully in [8]. Here we give a short overview, highlighting the division of labor between the DSS and PS.

2.1 Distribution Model

The distribution support provided by the DSS is to allow threads on different machines to transparently share language entities just as if the threads reside within the same process. Modulo failure and performance there is no difference between sharing within one OS-process and across a network. Similarly, and this may be more important, there is no difference when the distribution of threads changes. For example, given three threads A,B and C located on two OS-processes, there is no difference between the case when A and B are co-located on the same process on the one hand and when B and C are co-located on the same process on the other.

2.2 Sharing Language Entities

The Distribution Subsystem (DSS) is a middleware library, designed to provide distribution support for a programming systems [6]. Programming systems suitable for coupling to the DSS need to be referentially secure. This is, of course, good for security, but is also a key property needed to realize efficient distribution. Theses suitable languages do not allow for pointer arithmetic or any other language mechanism that permits a thread to gain access to a data structure without having a direct or indirect reference to it.

Distribution support is on the level of language entities/data structures, over an interface of abstract entities. Associated with an abstract entity type is a consistency model, e.g. sequential consistency for shared objects. The DSS provides one or more consistency protocols for each supported abstract entity type [8].

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

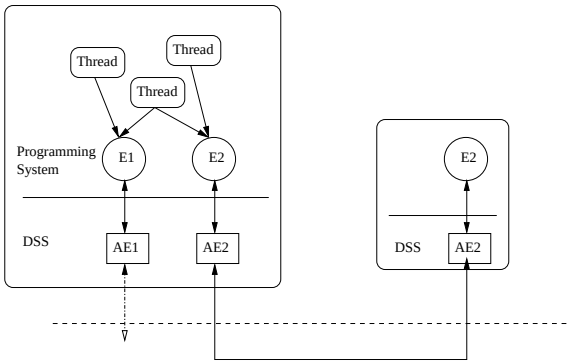

Figure 1: A programming system connect language entities to the abstract entity instances of the DSS and can then take full advantage of all features provided by the DSS. Entity E2 is shared between the two depicted processes, both running the DSS

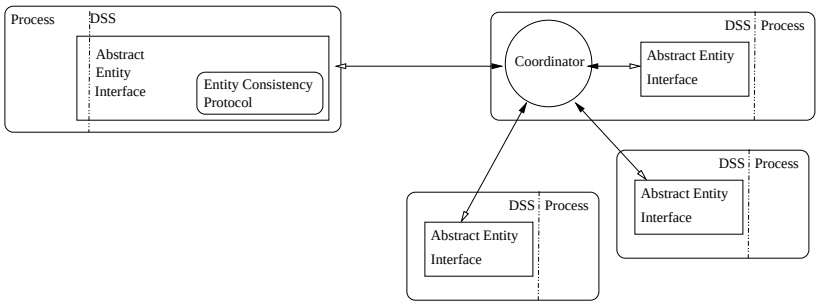

Figure 2: The coordination network and the per-process coordination proxy. The abstract entity instance is coupled to a coordination proxy connecting it to the coordination network. Note that in this example the coordination hub consists of a single coordinator located at one of the processes hosting a coordination proxy

Figure 1 depicts threads sharing entities (E1 and E2) locally and also with threads on other processes, through the abstract entity instances (AE1 and AE2).

Hereinafter, we refer to a process that executes the DSS as a *DSS-node*. Each DSS-node is assigned a globally unique identity. In addition, a DSS-node's identity is separate from its address which is important to overcome network asymmetry and achieve mobility[24].

For each shared language entity there is a coordination network. All DSS-nodes that share or have a reference to a given entity belong to the coordination network. In addition there may be one or more coordinator nodes, responsible for the necessary coordination to uphold the consistency model for the language entity. The nodes in a coordination network may or may not know of each other. The extent to which members of a coordination network know of each other depends on the entity type. Note that nodes that share the same entity can potentially interact with each other. All DSS-nodes know the coordinator(s) of all coordination networks that they are a member of. Coordinators may know none, some or all of the nodes in coordination network. Once again, this is dependent on the entity type.

Clearly a coordination network is a fine-grained concept, one for each shared entity.

A DSS-node may thus be a member of many coordination networks, one for each shared entity that the node is holding a reference to.

2.3 Division of Labor

The DSS provides distribution support for the programming system. In order to understand what is meant by support it is useful to recapitulate what the programming system should provide by itself.

The programming system provides the concepts of threads (or fine-grained processes), data structures and the associated operations. Two threads may share an entity within an OS-process. The programming system is responsible for maintaining the consistency of any type of data structure (including objects and code) according to the programming language semantics. In addition the programming system (possibly making use of operating system features) ensures that the processor is shared fairly between the threads according to the programming model.

In general, at any one time a thread references directly or indirectly only some of the language entities or data structures within the OS-process. This set, the reference set, is constantly changing. References may be lost as variables go out of scope and references may be gained by the action of other threads. An example of the latter is when one thread updates an instance variable in a shared entity, making the new value available to all other threads that share the same entity.

A central notion in the DSS that makes it virtually programming language independent is the notion of abstract language entity. There are numerous programming languages with numerous concrete language entities and operations, reflecting a wide range of semantics. Only some of the essential properties of language entities are needed for distribution support. One of the tasks when a programming system is coupled to the DSS is to classify concrete language entities and operations into abstract entities. The programming system must be able to distinguish between shared and local entities so that it can co-operate with the DSS whenever operations are attempted on shared entities. In addition the programming system must be able to marshal/unmarshal (serialize) language entities.

2.4 Bootstrapping

Without additional instrumentation referentially secure programming systems have the drawback of being difficult to bootstrap. In this model a newly created thread must be given an initial reference to some shared entity. Without this the thread can only work in isolation, not interacting at all with other threads. There is no way to connect computations or threads that are not already directly or indirectly connected via the transitive closure of the reference sets of all entities that are currently referenced.

In order to allow for more dynamicity the DSS offers an additional mechanism for bootstrapping. The DSS can, if so instructed, construct a ticket (a string) from an arbitrary language entity. This ticket can then, by some means outside the system of connected DSS-nodes, be given to another DSS-node. The recipient DSS-node will then have a reference to the original entity. Creating a ticket to an entity implicitly makes the entity shared even if the ticket is never actually used. As tickets may be freely replicated outside the system only a subset of the otherwise many available memory management strategies are applicable.

3 The Security Division of Labor

We shall now consider the division of labor as regards security between the DSS and the programming system.

On an abstract level, sufficient for the DSS model, we view the programming model as a model over threads and language entities. Threads have references to language entities and as some language entities are mutable threads may acquire references to entities that it previously did not know via the action of other threads. This is under the control of the programming system and the DSS is not involved at all.

On the programming system level the security model may have notions of users, rights, capabilities, etc. However at the DSS level with little loss of generality we restrict ourselves, at least for now, to the concepts of threads, language entities, and references. Users are modeled as threads and all threads that reference a certain language entity have essentially the same rights and privileges on it. Note, that this does not mean that the concepts of varying rights and capabilities cannot be modeled on the programming system level. For example, we have the difference between read versus read/write rights to a file. This can be modeled as two different entities, one for each type of right.

Thus the programming system is responsible for ensuring that when references (or rights, or capabilities) are passed from one thread (or user) to another that this in accordance with the security policy and restrictions that are desired.

What then is the responsibility of the DSS? The goal of the DSS is to ensure that given that the program and programming system are both correct that the invariants are not broken when the threads (or users) reside on different machines, or in different environments (i.e. moving from a closed LAN environment to the Internet).

The threats are:

Access A thread (user) should not be able to access/update an entity to which it has not been a given a reference at the programming system level. Here forgery of references as well as eavesdropping must be taken into account.

Destruction A thread should not be able to destroy an entity to which it has not been given a reference. By destruction we mean that other threads with legitimate references are denied access/update to the given entity.

Resource exhaustion A thread should not be able to exhaust system resources in such a way as to prevent other threads from making progress on their legitimate work. A denial-of-service attack is the classic example of resource exhaustion attack, but there are others. In general, if there is a model of fairness on the programming system level, this should be maintained in the distributed scenario. One thread should not be able to prevent another thread from making progress.

Denial-of-Service Attacks and the DSS Traditional Denial-of-Service attacks (DoS), massive communication in order to block out useful communication and computation, is an unsolved menace on the Internet of today, although there is work on overlay networks to prevent the effect of this kind of attacks[15]. The problem is on the level of the operating system. Thus a proper solution requires operating system support. The DSS cannot cope with this kind of attacks. A DoS attack in the context of this work can either be repeated initiation of the connection establishment protocol or non-useful interaction with a valid entity, i.e. entity operations with the cause of consuming memory or processor cycles.

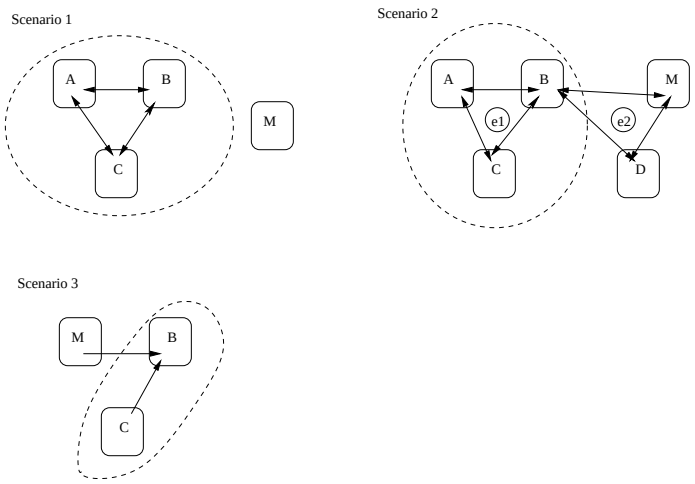

Figure 3: Three security scenarios. 1) Outsider attack 2) Indirect attack 3) Insider attack

4 The Three Security Scenarios

We now present the three main security threat scenarios. Here we consider the action that a malicious DSS-node might take. Note that we do not exclude that the malicious node has access to the DSS-implementation or source code.

The first scenario is the *outsider* scenario. We have a group of DSS-nodes sharing an entity. The attacker does not have legitimate reference to the entity. Moreover the attacker does not have even have a legitimate reason to connect to any node within the group, as the attacker shares no other entity with any of the nodes within the group.

The second scenario is the *indirect* scenario. We have a group of DSS-nodes sharing an entity. The attacker does not have a legitimate reference to the entity, just as in the first scenario. However the attacker does share other entities with some or all the members of the group. The attacker does, in other words, have a legitimate reason to connect to group nodes.

The third scenario is the *insider* scenario. Here the attacker is actually a member of a group of DSS-nodes that share an entity.

4.1 The Outsider Attack

The outsider attack is visualized in figure 3 scenario 1. A set of DSS-nodes (A, B, and C) share the same entity. The malicious node M has no legitimate reference to the entity nor does it share other entities with nodes A, B and C. However, we must take into account that M can monitor the traffic between nodes A, B and C, and that M can guess the IP-address of A, B and C. Note that the entity may have become shared between A, B and C via normal reference-passing (making use of another shared entity) or via the ticket mechanism.

The incorrect access problem is twofold. Node M should not by monitoring network traffic nor by fooling sites A, B or C be able to access or update the entity. We also have to assume that node M can masquerade his own identity, so that he may make many attempts to obtain access.

The two key ideas to the relatively straightforward solution to the incorrect access problem is encryption of communication and the (in practice) unguessable entity identifiers. Node M obtains no useful information from network monitoring. Although M can by guessing or knowing IP-addresses contacts nodes A, B and C, they quickly disconnect as M does not have the correct identifier for any shared entity that A, B, or C hold.

In this scenario the destruction problem is very similar to the access problem. If we can prevent node M from getting access to the entity we can also prevent node M from corrupting it. However, there is one slight difference. Node M may be able to indirectly destroy the entity by causing node A, B or C to crash. Nodes A, B and C do not know a priori that M is an impostor; this has to be established in the initial part of the dialog between the two nodes. We see that it is important that this part of the protocol needs to be robust to protocol violation.

The resource exhaustion problem in this scenario is a classic denial-of-service scenario. Although nodes A, B, and C can fairly quickly determine that M is an impostor, this determination still consumes some resources. If the attacks are frequent enough this may interfere with legitimate protocol operations involving nodes A, B and C.

4.2 The Indirect Attack

The indirect attack is visualized in figure 3, scenario 2. We have five DSS-nodes sharing two entities. Nodes A, B and C share entity **e1**. B, D and M share entity **e2**. Here we are considering the scenario where M is incorrectly allowed to access, destroy or exhaust the entity **e1** from the viewpoint of A, B and C.

Particularly when it comes to incorrect destruction and resource exhausting this opens up many more possibilities than M had in scenario 1. One way of destroying the entity **e1** is to cause node B to crash, which effects not only node B, but may also prevent nodes A and C from working with entity **e1**.

In this scenario there are some additional considerations on the implementation of programming system level unmarshaling routines. Unmarshaling of code and data must be both robust and referentially secure. The well-known problem of byte code verification is in this category.

The unmarshaler must be able to recognize corrupt data and code. In particularly the unmarshaler must not crash the entire OS-process. In the example, if node B receives corrupt data in accordance with operations on the shared entity **e2** it is ok that this destroys entity **e2** but this must not destroy entity **e1** as well, as would be the case if the entire process crashes.

A more subtle requirement on the programming system is that referentially security must be guaranteed by the runtime system, it is not enough to rely on the compiler. For example, it must not be possible to hand-code a byte code sequence associated with operation-shipping on entity **e2** so that other memory areas - in this case those associated with entity **e1** are actually addressed.

Finally, there are now some additional aspects to fairness aspects as regards both computation and memory, at least for some of the protocols. Node M may send marshaled representations of very large data structures, unexpectedly exhausting the virtual memory of node B.

We now consider this scenario from the perspective of the requirements on the DSS. In this scenario the access problem is almost the same as for the outsider scenario. The one additional consideration is that the legitimate communication according to the

protocol concerning entity **e2** should not provide any additional clues as to the identity of the **e1**.

The destruction and resource exhaustion problems in this scenario are more severe. The entire consistency protocol for entity **e2** must now be robust to attack with a view to crashing the other nodes. This differs from the outsider scenario where only the opening sequence of protocol messages needed to be robust.

Aside from denial-of-service attacks there are some additional potential resource exhaustion problems. The rationale of the attack is to cause node B to use all of its time and resources dealing with entity **e2** starving proper operation of the protocol for entity **e1** in node B.

4.3 The Insider Attack

The insider attack is visualized in figure3 scenario 3. Here node M has a legitimate reference to a shared entity. Note that in this scenario the access problem is not relevant, as the malicious node has a legitimate reference. However, the destruction problem remains, in that the malicious node may or may not be able to destroy the entity, so as to make it unusable for other nodes sharing the entity. In the figure, node M may or may not be able to destroy the entity from the viewpoint of node C.

While the insider and indirect attack scenarios are more or less independent of the particular entity type and the associated consistency protocol this is not true of the insider attack scenario.

First, for some protocols, given that one participating node is malicious there is nothing that can be done. A good example of such a protocol is the migratory mutable protocol, where the mutable entity is shipped to the DSS-node that is working on it, so that the operation may be performed there. Not only may the malicious node hold the mutable data forever, denying it to all others, it may arbitrarily corrupt the data, making it unusable to others.

Other protocols, like the asynchronous message-sending protocol or stationary object (RMI or RPC-like) are amenable to security instrumentation. The reason is that these protocols are essentially asymmetric in the capability associated with entity references. In the stationary object the DSS-node holding the object state is privileges. If that node is malicious then nothing can be done to prevent malicious access of the data. However, if the malicious node is only holding a reference and has no direct access to the state then protection is possible.

5 DSS Internals

Central in the DSS is the consistency protocol framework. This framework enables simplified development of protocols as indicated by the large suite of efficient protocols provided by the DSS [6]. The key component in this framework is an efficient and expressive inter-process service. As shown in Figure 4, the DSS is internally divided into two layers:

Protocol layer This layer implements the consistency protocols that coordinates accesses to the shared entity. The protocols are divided into three subcomponents or strategies. Two of them are independent of entity type. First the memory management strategy responsible for distributed garbage-collection. Second the

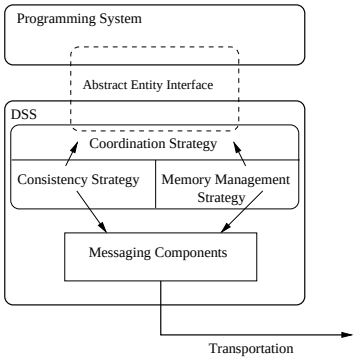

Figure 4: The structure of the DSS middleware library. Within the DSS, the protocol layer consists of three sub components/strategies all using the messaging layer to communicate with other DSS-nodes

coordination strategy defines the way in which consistency coordination or arbitration is achieved. The coordinator may be stationary or mobile, single or redundant. Finally we have the consistency strategy. This is dependent on abstract entity type. This protocol ensures that entity consistency is maintained.

Messaging layer Inter-process interaction tasks, e.g. messaging abstractions and fault detection, is realized in this layer. It provides the abstractions the protocol layer use for messaging and expressing locality.

All the processes that have a reference to a shared entity together with one (or more) coordinators form a network, the *coordination network*. The data structure is called an *abstract entity instance*, and in the course of protocol operation abstract entity instances will send messages to one another. This is depicted in Figure 2.

At any point in time a DSS-node may know other DSS-nodes; these nodes are referred to as the *known set*. During the course of computation, references to DSS-nodes are passed between DSS-nodes in a diffusion manor, thus the *known set* changes. At any one time a DSS-node needs to communicate with a subset of the *known set*, this subset is also constantly changing. It is perfectly possible that a DSS-node will never communicate with a given node in the *known set*. The known set is implicitly controlled by the entity consistency protocols.

5.1 Messaging Layer

The DSS-node representation, the *DSite*, is a composition of an identity and an address. A DSite express locality, offers a messaging interface for protocols and expresses accessibility of the remote process, i.e. detects network perturbations and classifies them into fault states.

The DSite is internally divided into different components, realizing different tasks and also customizable for different application needs[7]. The DSite representation is dynamic and depends on the use. If communication is needed the DSite is said to be *active* and is comprised of five data structures, depicted in Figure 5. First, the two left-most data structures manage identity (DssSite) and address (CscSite). Second, a single (Session) data structure manages connection negotiation and reliable message sending. Finally, actual transportation of a message is done by the Transport and Channel

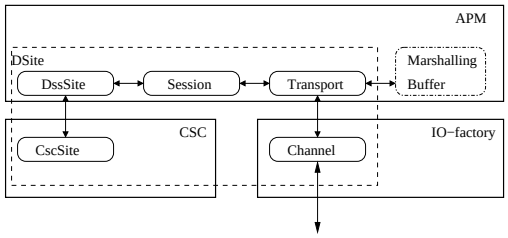

Figure 5: The complete runtime representation of an active DSite. The five runtime data structures that constitutes a DSite resides in all three modules of the DSS. Each data structure realize a specific task domain, e.g. transportation, addressing and transmission control.

object. The Channel object is an abstraction of a communication channel, i.e. sockets. The Transport object is an interface to different types of Channel. Marshaling of messages into binary format for transportation is done by the Transport object. The last data structure, conceptually not a part of the DSite runtime representation, is the marshaling buffer, used to store serialized messages waiting for transportation.

When a new, not currently present DSite representation, is received at a DSS-node it is added to the known set. It is not until a protocol instance uses it for communication that the DSS-node establishes a connection to the represented DSS-node. Connections are thus established on a by-need basis, according to the address part of the DSite. When no communication is needed the channel is eventually dropped.

The addressing schema of the DSS is a customizable component, allowing for change of both run-time address and addressing schema. Effectively this means that the serialized representation of a DSite is dynamic, consisting of a static identity part and a (possibly) dynamic address part. During a distributed computation DSS-nodes might physically move or logically change address, e.g. when running on a mobile device.

6 Making the Basic DSS Services Secure

The DSS is designed around the concept of references. This is true on the consistency protocol layer and on the messaging layer. By holding a reference to a process in the form of a DSite, a connection can be established to the process. The same holds for entity consistency protocols; a reference to a coordination network gives full access to the shared data structure the coordination network coordinates. Due to the expected diffusion behavior, the garbage collection protocol and the fact that a reference is a capability to enter a coordination network, references are not stored in a centralized repository but received and managed locally on a DSS-node. Consequently, there is no notion of which processes that has received a reference to a process or to a coordination network.

The reference based schema is elegant and efficient. A reference to a shared data structure can be passed between two processes without involving third party processes. However, by forging references, a process could be granted access to both processes and shared data structures it should not have access to – this is the rationale for capability systems in general, designation and authority comes hand in hand.

6.1 Requirements on a Secure DSS

We have identified the following security requirements in the DSS.

Unforgeable DSite References In order to establish contact with a DSS-node, a correct DSite instance is required. Making DSite references unforgeable prevent processes from connecting to a DSS-node without holding a correct reference to that process, i.e. knowing or guessing the physical address is not enough. Furthermore, a correct reference to a given DSS-node can only be created by a DSS-node itself. Other nodes learn of a DSS-node (in the form of a DSite) by reception of DSite references. Unforgeable DSite references are required in scenario one and two. In scenario one the attacker needs a valid DSite reference to connect to one of the other nodes and in scenario two the two untrusted nodes (M and F) is not aware of nodes A and C, i.e. B effectively becomes a gateway for communication.

Robust Connection Establishment The ability to connect to a DSS-node must be restricted to prevent non-trusted nodes from causing damage a DSS-node. Only a DSS-node that has the capability to connect to a DSS-node should be able do that. A modified connection protocol must withstand buffer underflow/overflow attacks and repel attackers quickly in order to minimize effects of Denial-of-Service attacks. This condition can be mapped to scenario one where an attacker is kept out side of the network of trusted nodes.

Encrypted Channels Any observed communication between two parties should never reveal identity or addressing or other sensitive data, except the obvious fact that they are communicating. Similar to the previous requirements this apply to both scenario one and two, if all communication remains hidden a malicious node can not contact any of the trusted nodes or read data in transit.

Each protocol type within the DSS has a set of message types. Thus means that although messages in general are different, patterns can be detected and to conceal them, and prevent an eavesdropper from deducing what is sent, cipher encryption must support a feedback mechanism, e.g. Chained Block Cipher (CBC). Furthermore, two communicating DSS-nodes must be able to detect if someone has tampered with the channel – i.e. check data integrity – and take appropriate actions.

Unforgeable DSite Addresses A DSS-node has the ability to change its contact information (e.g. its IP and port number in a TCP based network). Information regarding a nodes address change diffuses out from the node itself to the other nodes in a network of a distributed computation. A possible attack to a distributed computation is to insert an invalid address for a particular node, and thus making it impossible to contact the node. Thus, address changes for a particular node can only be issued by the node itself. This requirement can be mapped to scenario two, from destruction point of view. If a malicious node M can convince another node D that a third node B, which contains a coordinator for entity e, has moved to arbitrary location then the entity becomes useless for D.

Unforgeable Coordination Network References Similar to the DSites, coordination networks are accessed using references. A reference to a coordination network should only be constructed by reception of a reference to the coordination network. These identities should be verifiable, i.e. it should be possible to tell

whether an identity has existed or not, thus rendering it possible to detect continuous searching attempts for entities.

This requirement maps to scenario two where the malicious node M is unable to obtain entity **e1**.

7 Design

The design of the security enhancements for the DSS is governed by the requirement to affect as little as possible of the current designs favorable properties and features. Only as a last resort should something be removed or seriously crippled. Furthermore the DSS is constantly extended and changed to add more functionality and improve performance. All in all the design and implementation of security features should be isolated and transparent to the maximum extent for the abstractions that use them.

The use of the widely known Secure Socket Layer (SSL)[1] for communication and addressing was ruled out for several reasons. First the DSS offers internal routing and needs to set up indirect end-point communication using other existing channels while not exposing traffic to the intermediary nodes (not possible with SSL only). Second, addressing is highly dynamic and not directly bound to sockets, so the DSS can not rely on TCP/IP addressing as SSL does. Last, the asymmetric keys are used for other things than just session key establishment, e.g. signing internal abstract data like capabilities, DSites etc and should be made available through the DSites together with signing/verifying functions.

For an eavesdropper to obtain the clear text transmissions of the channel there are two approaches: either guess the session key of the channel or retrieve the ticket used to connect then obtain the identity-key pair. Both approaches are made difficult enough to prevent intrusion.

7.1 DSites - Identity and Address Protection

The task for a DSite is to provide identification and establish connections when needed. The DSite representing the local DSS-node is used as a tool to prove origin and intent, e.g. to prove creation of references, verify and protect address representations and sign references. To comply with these tasks and requirements the asymmetric RSA[20] key algorithm is used by the DSite. This algorithm is fairly simple to implement and provides adequate strength. The key size is defined at compile time for the DSS library and should be set to guarantee[17] enough safety for an ongoing computation.

Each local DSite is paired with a private key used for signing. The matching public key is used as the identity of the DSite. This identity, and henceforth the key pair, is fixed during the entire lifetime of the DSS-node. Having the public key as identity is preferable for two reasons: first the public key anyway has to be distributed with the DSite for the completeness of the connection protocol, second the keys are, and has to be; unique which is the case for the identity too. The address part of the DSite is dynamic and is paired with a counter representing the address version. Every time a CscSite instructs its DssSite that its address has changed the counter is increased. When a DSite is spread around the network of DSS-nodes, they can update DSite representations with the latest (seen) address info.

Using a public key as identity prevents masquerading; a malicious DSS-node must have the private key to sign the hash. This schema would of course fail if some DSS-node would find out the private key, but that is a matter of key strength. Longer keys

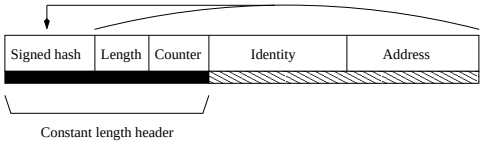

Figure 6: The layout of the marshaled DSite representation. The identity and address part is variable length. The header, consisting of the length of the representation (in bytes), address counter and a signed hash of the other elements

would improve security at the cost of additional computation, both when applying the keys and when generating the big prime numbers[18, 14] used by them.

DSites monitor its channel to detect if a fault has occurred. This information is propagated to the coordination networks so that they can take appropriate actions. This state information is extended to cover security, thus a DSite can also be, apart from regular fault states, regarded *suspected malicious* or *malicious*, which is propagated up to the communication layer similar to regular faults.

To improve performance during marshaling a pre-serialized representation is stored with every DSite. Figure 6 shows the layout of the serialized representation. The complete representation is a composition of the serialized representation of the public key/identity and the address. The address counter is stored in the header together with the length of the serialized representation. A MD5 hash is calculated on the above composition and then signed with the private key. This signature is then added to the complete representation.

To avoid illegally structured DSites from appearing on the network of DSS-nodes all marshaled representations are verified upon first importation. This schema prevents an intruder from spreading fake DSite representations.

When a DSite representation enters a new DSS node the identity is compared against other stored DSites to identify duplicates of the same DSite[2] according to the following tests: :

Identity exists with same signature. The newly imported representation is discarded and the old one used.

Identity exists but signature is different. The counter versions are matched; if older the newly imported representation is discarded. If newer the validity of the signed hash is checked and if correct the CscSite is updated with the new information and the DssSite with a new marshaled representation.

Identity does not exist. The signed hash is verified, for mutual trust, to avoid importation of obviously false DSites. If invalid the exporting DSS-node is also disconnected and marked as malicious since it should have detected the same during its importation of the Site. Similar to how failures are handled the DSS informs the CSC about the faulty site and discards the newly unmarshaled representation.

[2]DSites are governed by the at-most-one-copy property, independently of how many times a DSite is imported only one representation is stored. This facilitates updates of address information and avoids memory explosion.

7.2 Connection Establishment and Channels

A connection between two DSS-nodes is established, if none exists, when messages are to be delivered between the DSS-nodes. During connection establishment the Session object negotiate settings, e.g. buffer sizes, ping intervals, encryption keys.

Message serialization is a cooperate task between the DSS and the CSC or the PS, initiated by the Session object when space is available for serialization and data can be transported. Serialization depends on the transportation type (e.g. stream, packet-based or virtual circuit) and the connected channel. However, after a message has been serialized it is always placed on a transportation buffer while waiting for actual delivery. To make encrypted channels transparent from the transportation type, since there are potentially many, encryption functionality is placed on the buffer controller. If a secure channel is requested during connection negotiation then the ordinary buffer controller is replaced with an encryption-enabled controller, fed with parameters from the connection establishment. Both the replacement and the usage of an encryption-enabled controller are entirely transparent for the various transportation objects.

The previous connection establishment protocol is described in [7]. In short the initiating side asks the CSC for a direct channel or route to the destination and when given transportation means simple clear-text verification is done. The new protocol uses the public key of the DSite to protect the entire negotiation from eavesdropping. The identities of any participant is never revealed in clear text and the contacted DSS-node assumes, and verifies, that the initiating side knows the identity/key, hence an observer cannot establish a connection to anyone of the participants; knowing the identity becomes a capability to connect to the network of DSS-nodes.

Outlined below is the connection negotiation procedures, the initiating side are denoted (I) while the contacted side is denoted (C). The initiating side requests a channel to the contacted side and when a channel is established and eventually handed to the session objects negotiations start. To prevent replaying of negotiation messages nonces are used as capabilities to verify correct state transitions during negotiation. Message element extraction is protected from buffer underflow and if the protocol encounters invalid data or buffer problems the channel is immediately dropped and the other side is marked as possibly malicious.

1. (C) creates a session object that is handed the newly created transport object. (C) then sends a presentation message containing a nonce for verification (capability for further contacts). The message is encrypted using the private key of the local DSite. This message verifies that (I) know the identity of (C).

2. (I) receives the presentation message and decrypts and extracts the nonce. If (I) didn't have the identity of (C) then the message would be unreadable. (I) then creates a presentation message with its identity. The received nonce, a new nonce and requests for security and session parameters (e.g. proposed timeouts and proposed session keys) are encrypted with (I)s private key and stored in the message, i.e. I is proving that he really is (I). The complete message is encrypted using the identity-key of (C).

3. (C) receives (I)s presentation and verifies the nonces. If (C) agrees on the proposed security/session settings the buffer controller for the incoming buffer is security enabled. At this stage simultaneous connections are detected and resolved. (C) creates a message including security and session responses and encrypts it using the private key of (C).

4. (I) receives the answer message and if encrypted channels are used both the incoming and the outgoing channel are changed to encrypt. A final connection-negotiation message is then sent to (I) before other messages can be serialized.

5. (C) receives the last negotiation message and can change its outgoing channel to be encrypted if that is what was decided. Afterwards all queued messages can be delivered.

The encryption technique used for the DSS is a block cipher in CBC mode, effectively hiding reappearing messages. The particular block cipher can customized. Using a simple interface offering basic operations e.g. encrypt, decrypt, key setup, the algorithm of choice can be selected and used. The encryption algorithm is a matter of taste and trade-off between performance, strength and key set up speed. The schema simply ensures that future improved algorithms can be incorporated into the DSS. In the current implementation the blowfish algorithm[21] was selected mainly due to the simple implementation and favorable performance[22]. Note that any other block cipher algorithm, like the well-known 3DES, IDEA or AES (Rijndael[3]), can be used.

The crypto buffer-controller takes serialized data and encrypts it into frames of at most a given (negotiated) size. An Adler32[5] checksum of the plain text is calculated and attached, together with the frame length value, to the frame header.

A fully transmitted frame with correct checksum is given to the marshaler that deserializes data into Message Containers, while performing some basic verification of data, e.g. correctness of size specifications for data types, valid DSite representations and complete lists. In case the checksum or length mismatch the transport object is instructed to force a close of the channel. In case communication is still required a new session has to be negotiated. The transportation object is totally unaware of frame splitting and just transmits bytes as usual, disregarding frame boundaries.

7.3 Unforgeable Entity Identifiers

Shared abstract entities have their local representative in programming system space coupled to an abstract entity instance in the DSS. The protocol instances belong to the corresponding coordination network. [6]. This network is identified through a globally unique entity identifier or GUId, thus the GUId effectively creates a capability to access the coordination network. The realization of basic unforgeable references depends on this identifier.

The representation of a GUId is consists of two data structures:

1. The DSite of the creating DSS-node

2. A counter and some control data which is signed with the private key of the creating DSite from above.

.

Knowledge of the creator DSite together with the sizable and, per DSS-node, unique signature, is enough to pin-point a coordination network.

The GUId is created together with the rest of the data structures for a new coordination network. The GUId is distributed with every reference to the entity. When a protocol instance wants to send a message it creates a message container with enough space for data, i.e. GUId and message type, for it to be delivered correctly.

On the receiving side the GUId is matched against GUIds of local coordination proxies and coordinators. If the GUId exists the message is delivered to the correct abstract entity instance, else it is silently dismissed.

8 Evaluation

Testing the implementation was carried out in order to verify the design. Evaluation of the RSA public key algorithm and the cipher algorithm was not carried out since in this context, they are state-of-the-art encryption algorithms. Three implementation tests were performed, covering the security additions to the DSS:

Connection negotiation. Flaws in the connection establishment protocol were checked for by feeding the channels with inaccurate data as well as both oversize and undersized inputs. DSite representations were also manipulated in order to verify that any deviation from the specified format was detected.

Detecting channel perturbation. Channel communication was tested by inserting, removing and changing random bytes. This tested the buffer controller design and verified that problems regarding channels were correctly detected and the channel closed in case of errors.

Detecting forged messages. A test for invalid references to coordination networks was performed. References were generated and inserted into message containers, these were detected at the other side and the entire message discarded. Due to the dynamics of the Message Container and the incremental marshaling of the DSS the entire message has to be deserialized before it can be discarded. Preferably a message should be discarded directly when an invalid reference is detected and the buffered simply cleared while the data is completing. This is something that should be investigated in later stages of securing the DSS.

9 Capabilities as a Security Mechanism

Internally the reference and diffusion model of the DSS is inherently oriented towards the capability access control mechanism. Capabilities are in nature non-discretionary, suitable for the DSS which offers diffusion based distribution model and provides a non-discretionary distribution model. Capabilities have been identified as an access control mechanism for many years [4]. Work on capability mechanisms includes areas as operating systems, programming and object systems and distributed applications, which has shown it to be a general and plausible security mechanism [9, 12, 23, 13, 19, 11, 10].

Another mechanism for access control is access control lists (ACL). ACLs are discretionary and provide simpler means for discretionary control, e.g. revoking or extending user rights. However, for a reference diffusion model a capability system, where delegation is trivial and cost-effective, becomes more attractive.

Important to note is that there is nothing preventing the DSS from being connected to access control list based programming applications/systems. One possible programming model (for ACL) is to make use of two abstract entities: one representing the resource and the other user identity. Both of these are shared and needs to be protected from forgery by the DSS. Associated with the resource is a mutable access control

list. Moreover, the referential secure model is a powerful tool for implementing both capability and ACL based models on the programming system level.

10 Work Status

Focus has been on completing securing the most primitive mechanisms in the DSS; secure channels, unforgable DSite references and unforgeable entity references. These services are implemented in the DSS and have undertaken a first evaluation. All the naming schemas in the DSS are based on pairs of DSites and integer values. Unforgable DSite references are thus the base requirement for realizing unforgable entity identifiers.

Less attention has been devoted to choice of encryption algorithms. To simplify the development, and increase the chance of third party usage we have only used algorithms publicly available. The component oriented design of the security services makes replacement of particular algorithms feasible at a later stage, an interface through which adapted algorithms can be plugged in, is offered to DSS users.

11 Future Work & Discussion

We have in this document described three scenarios we expect the DSS to handle, in conjunction with a properly instrumented programming system. To fully handle the two latter scenarios in Figure 3 – to the greatest possible extent – we plan to improve several aspects of the DSS.

11.1 Protocol Robustification

If we consider the indirect threat scenario, we cannot, in general, prevent a malicious node from destroying the entity it has a reference to. In general, the protocols rely upon that the nodes cooperate. However this destruction (or resource exhaustion) must not spill over to other entities. The process-to-process interaction protocols must be made robust, i.e. there must be no way to crash a DSS-node using knowledge of the implementation.

11.2 Denial-of-Service

DoS attacks can be applied to several layers of the DSS. From pure, externally handled, socket connections to more serious consistency protocol interaction that claims more resources. Preventing attacks on sockets is easier to handle for the DSS since attackers are repelled quickly (they represent scenario one in the figure).

For instance, when a message to an entity is received that does not match any of the locally existing (within the DSS) entity instances the message is silently dismissed. No actions against the sending DSS-node is taken due the distributed garbage collection (DGC) schema of the DSS. The two reasons are: first the receiver cannot be certain that the GUId has not existed earlier or that the sender has not received a received a reference from a malicious site that kept the reference while a garbage collection was performed and start spreading it afterward. A solution could be to have verified references as soon they are imported, but this is probably extremely costly in terms of extra messages and synchronization.

The example is just one of many possible security holes in the DSS regarding denial-of-service and they must be treated in as far as possible or practical.

11.3 Trust Model

Offering a trust model for the DSS may be needed for two reasons. Firstly, a trust model may allow for necessary or useful optimization among trusted nodes. Secondly a trust model may be useful programming aid.

Implementing protocol robustification can – but might not – impose a significant execution overhead. If DSS-nodes can be trusted then this may be avoided.

On the programming system level the application developer is, in general, given a choice of consistency protocols to choose from for a given language entity. The choice governs only non-functional aspects, but may be important for tuning the application for performance. However, for some application the most efficient conflicts with the most secure. The trust model can be used to guide and/or restrict the choice of protocol.

11.4 Capability Model over Abstract Entities

The DSS model is based on the notion of abstract entities and abstract operations. For instance, sequentially consistent mutables can be updated or accessed. It may be advantageous to distinguish between the update capability and the access capability in the DSS. We want to investigate the potential of a capability model based on abstract entities and abstract operations supported in the DSS.

11.5 Programming System Interaction

Currently, messages that do not pass the security inspection are quietly dropped. It may be useful to propagate attempted security breaches to the programming system level. We need to investigate how best to expose this information to the programming system level. Also, it is probably important that if the security provisioning attempts to identify the misbehaving node that this identification is also secure. If not, a new type of destruction threat becomes possible, when a malicious node convinces other nodes that a, in fact well-behaved, node is misbehaving.

12 Conclusion

The work covered in this report represents a first step towards a security enabled DSS, from the aspect of the three scenarios discussed in section 4.

These scenarios exemplify the model of threats that the DSS will ultimately handle. A design of security extension that provides protection against some of the threats of these scenarios is also presented. The design offers encryption on channels between processes, but this is made optional, which is useful from a performance perspective when processes are trusted. Furthermore the component-oriented design allows the specific encryption algorithm to be replaced.

A secure and verifiable process identifier has been designed. This replaces the original identifier of the DSS, and supports the reference diffusion model of the DSS in untrusted environments. This identifier is designed with asymmetric key pairs. Based partly on these process identifiers are references to shared data structures made unforgeable.

We believe that the model provides a generic service for security, much in the same way as the DSS provides a generic service for shared data structures. The current implementation provides some of what is needed to ensure the model and work is ongoing to provide for the rest.

References

[1] P. Karlton A.O. Freier and P.C. Kocher. The ssl protocol: Version 3.0, March 1996. Available at http://home.netscape.com/eng/ssl3/ssl-toc.html.

[2] Mozart Consortium. http://www.mozart-oz.org, December 2002.

[3] J. Daemen and V. Rijmen. Rijndael, the advanced encryption standard. *Dr. Dobb's Journal*, 26(3):137–139, march 2001.

[4] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, MIT, 1965.

[5] L. Deutsch. Zlib compressed data format specification version 3. Internet RFC 1950.

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

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

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

[9] Li Gong. A secure identity-based capability system. In *IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[10] D. Hagimont and N. De Palma. Non-functional capability-based access control in the java environment. http://citeseer.nj.nec.com/537978.html.

[11] Daniel Hagimont and Leila Ismail. A protection scheme for mobile agents on java. In *Mobile Computing and Networking*, pages 215–222, 1997.

[12] N. Hardy. The keykos architecture. *ACM Operating Systems Review*, pages 8–25, September 1985.

[13] Secure multi-user distributed applications - waterken inc., November 2003. http://www.waterken.com.

[14] Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient generation of prime numbers. *Lecture Notes in Computer Science*, 1965:340+, 2001.

[15] A. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *Proceedings of ACM SIGCOMM 2002*, August 2002.

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

[17] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293, 2001.

[18] Ueli Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, 1995.

[19] M.S. Miller and J.S. Shapiro. Paradigm regained: Abstraction mechanism for access control. In 8^{th} *Asian Computing Conference*, Dec. 2003. To appear.

[20] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, MIT, 1977.

[21] B. Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). *Lecture Notes in Computer Science*, 809:191–204, 1994.

[22] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1995.

[23] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[24] A. Snoeren, H. Balakrishnan, and M. Kaashoek. Reconsidering internet mobility. In 8^{th} *Workshop on Hot Topics in Operating Systems*, May 2001.