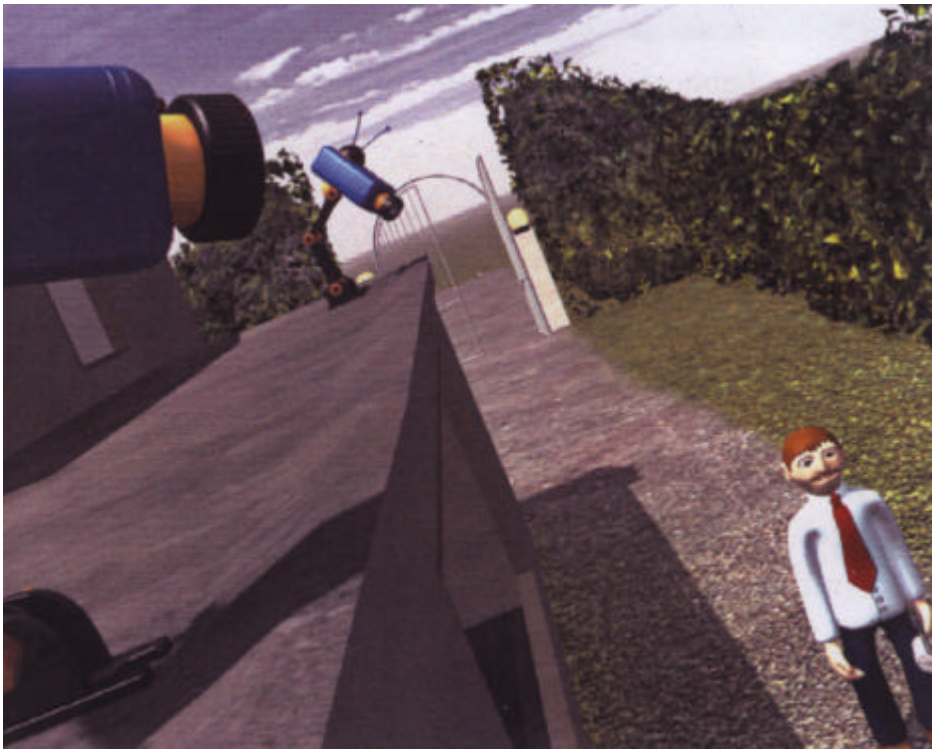
NID versus Splice



A Comparative Study of two Distributed Computing Platforms

Niels Mol
Universiteit van Amsterdam
April 2002

NID versus Splice

a comparative study of two distributed computing platforms

written by
Niels Mol

April 2002

for obtaining the title Master of Science
in the field of Computer Science
specialization Intelligent Autonomous Systems
at the Faculty of Science of the Universiteit van Amsterdam

under supervision of
Marinus Maris and Ben Kröse

keywords:
NID, Jini, Splice, distributed computing, surveillance, communicating devices

Abstract

At the Universiteit van Amsterdam we are developing a distributed intelligent surveillance system. We want this system to track moving objects across multiple cameras. This requires a distributed computing platform that handles the information exchange between the cameras. In this thesis we compare two systems that can perform this task, namely NID and Splice.
NID uses Java's Jini protocols for creating connections between devices and Java's RMI protocol for communicating data between devices. Splice is a shared data space system. Devices communicate via the shared data space, and data is distributed by a collection of standard processes.
We identify the most important issues for our application and discuss how NID and Splice handle each of these issues. To compare the performance of both systems we performed two experiments. The first experiment measured data transfers of different sizes. The second experiment simulated the communication pattern of our application with a varying number of (simulated) cameras.
The outcome of this study is that NID shows better adjustability, flexibility and robustness. Splice has a better mechanism for dealing with persistent data and shows better real-time performance and scalability. Since real-time performance and scalability are essential for our application, Splice is more suitable for exchanging data between the cameras in our system.

Contents

Chapter 1
Introduction

1.1 Background

Safety is a hot topic in today's society. The use of cameras to ensure safety is becoming more and more common practice. Normally this requires that a human operator monitors the video images from a number of cameras. By embedding the cameras with a certain level of intelligence and letting them cooperate, the cameras could do the monitoring themselves. At the Universiteit van Amsterdam (UvA) we are developing such a distributed intelligent surveillance system as part of the *Big Brother* project.
We want our system to track moving objects across multiple cameras. The first step is detecting a moving object in a video sequence that is generated by a camera. Therefore we use a background subtraction technique. This technique separates moving objects from the static background.
The second step is tracking the detected objects through the video sequence. This is done by comparing features of an object in one video image with features of objects in the next video image. In that way, objects can be tracked by a single camera as long as they are in the field of view of the same camera.
When an object disappears from the field of view of one camera and enters the field of view of another camera, we want to maintain tracking. Therefore the cameras have to exchange information about the objects. This requires a distributed computing platform that handles the information exchange between the cameras. This thesis concentrates on that particular part of the system.
Designing a distributed system is not a trivial task because there are many different aspects involved. The kind of application and its requirements will strongly influence the design of the underlying computing platform. Fortunately, there exist already a large number of such platforms, so we don't have to design one from scratch. In this thesis we will compare two operational platforms that we are familiar with at the UvA, namely *Networked Intelligent Devices (NID)* [12] and *Splice* [1] [2]. The result of this comparison will form the basis of selecting one of these two platforms for the distributed intelligent surveillance system that we are developing.

1.2 Framework application

If we want to make a detailed comparison, we first need to decide what we are comparing exactly. Therefore we present a framework application in order to define a clear frame-of-reference. In the following chapters we will describe how this framework would be implemented using NID or Splice. This will allow us to compare the two platforms.
The framework application consists of a number of interconnected devices. Each device consists of two parts: a camera and a computer. The computer processes the video-sequence generated by the camera. It tries to detect moving objects in the camera's field of view. Data about the moving objects is communicated to other devices in the system. When an object disappears from one camera's field of view and enters another camera's field of view, the other device should be able to decide whether it is the same object. The devices only keep

track of the current location of an object. They don't trace the path that the object has followed.
To limit the amount of data that is sent across the network, the devices only send object information to devices nearby. This is viable because devices nearby are more likely to detect the object than devices far away.
The devices are monitored remotely. An operator can see where the cameras are located, and where the detected objects are. The operator can also make adjustments to the devices remotely. He or she can pan or tilt a camera for example, or let a camera zoom in.
The application is easily expandable. New devices can be added without having to reconfigure the existing ones. This means that a new device contacts the existing devices, and is integrated into the network automatically.

1.3 Related work

This section describes a number of studies that are in some way related to the study presented in this thesis.

Wells et al. compared three shared data space systems [18]. These systems are JavaSpaces from Sun, IBM's Tspaces and Wells' own system, which is called eLinda. All three systems are based on the Linda programming model. Since Splice bears strong resemblance to Linda, the three mentioned systems are comparable to Splice. Another interesting point is that JavaSpaces uses Jini technology, just like NID does. All three systems are implemented in Java and use a central data server. Experiments with a ray-tracing application distributed over a small number of computers showed only small differences in performance.

Mohindra and Ramachandran did a comparative study of distributed shared memory system design issues [13]. They tried to identify a set of issues that define an efficient implementation. The issues that they consider most important are integration of distributed memory with virtual memory management, granularity of computation and data, choice of memory model and choice of coherency protocol.

Both studies compare different implementations of the same concept, whereas our study compares two implementations of different concepts. However, it does show us how a comparison could be done.

López de Ipiña developed a system that shows some similarities to our framework application. His TRIP system [10] is a distributed vision based sensor system, which uses visual markers (stickers with a barcode) to identify objects with a camera. Information about the detected objects is communicated to a central server that manages this information.

There are also a number of more general studies about distributed systems that are relevant to our framework application [5] [6] [8] [9] [11]. These studies show which issues are important for distributed systems. From these issues we select the issues that are relevant for a network of communicating devices.

1.4 Thesis outline

In this chapter we gave a brief description of the Big Brother project and the goal of this study. We also described a framework application that will be used for the comparison and we discussed some work from other authors that is related to this study.
The next two chapters will introduce the two systems that we are going to compare. Chapter 2 introduces the NID concept and chapter 3 introduces Splice. The crucial issues in the design of our framework application are discussed in chapter 4. For each issue we will discuss how NID and Splice handle it, and we will try to decide which of the two systems handles it better. For a further comparison we have created two experiments. These are described in chapter 5. Our findings in chapter 4 and 5 will be discussed in chapter 6. A final conclusion will be presented in chapter 7.

Chapter 2
The NID concept

This chapter describes the NID concept, which is developed to create a network of devices that cooperate to perform a certain task. It uses Jini [16] for creating connections between devices and RMI [15] for communicating data between devices.

2.1 Introduction

TNO-FEL developed the NID concept for creating a network of interconnected devices that cooperate to perform a certain task. NID stands for Networked Intelligent Device. Typically, a NID has a hardware and a software part. The hardware part consists of a sensor or actuator, a network interface and a computation part. The software consists of a part that processes the signal from the sensor/actuator, a part that manages the communication and a part that performs the intelligent tasks. We will concentrate on the software part that is responsible for all communication.
The devices in a NID system perform a number of tasks autonomously, such as analyzing the data it gets from the sensor. Devices communicate the results from their analysis to other devices in the system. On the basis of the shared information, they then cooperate to make an elaborated decision about the actions that have to be taken.
Each device is capable of functioning correctly on its own. In other words, a device does not rely on other devices. This ensures that the system keeps on functioning if a device breaks down.
The interconnection pattern of a NID system and the behavior of the individual devices can be configured at runtime. The connections between devices are established with Java's Jini technology. Communication with devices and between devices is done via Java RMI. The next two sections will describe Jini and RMI.

2.2 Jini

The Jini Technology Architecture is an addition to the Java programming language, which facilitates the use of distributed services. The *service* is the most important concept of Jini. A service can be hardware, software or a combination. Examples of services are printing a document, converting a file from one format to another, storing information, etc. Services can be used by a person, a program or another Jini service. A collection of services together makes up a Jini system. Such a system is also called a *federation* or a *djinn*.

The goal of the Jini Technology Architecture is to create a flexible, easy administered network of distributed services. The Jini Programming Model makes it easy for users and programs to make use of services and share services.
The Java Application Environment enables code and data to be run on any location with a Java Virtual Machine (JVM). Jini makes use of this, so users can change their network location but still access services in the same way.
The Jini infrastructure creates a dynamic network to which services can easily and automatically connect and disconnect from. It does not require any network configuration.

All of the entities involved in a Jini system may reside on different physical locations. The only requirements are that there is an IP network connection between the locations, and that there is a JVM running on every location.

The glue that holds a collection of services together is the *lookup service*. The lookup service acts as a marketplace for offering and finding services. When a new service is created or started up, it first has to find one or more lookup services. This is called the *discovery* process (see figure 2.1). After this the service has to register with one or more of the lookup services it has just found. This is called the *join* process. When the service is registered, it can be found by other entities (persons, programs or other services) which have discovered one of the lookup services the service is registered at.
An entity can get access to a service through the lookup service. This is called a *lookup*. The entity asks the lookup service for a certain type of service, and the lookup service returns references to the matching services. The entity can then access the service directly through the obtained reference.

So far, we haven't mentioned devices in Jini; we only mentioned services. In a NID system, every device offers one or more services to other devices. A device registers its services at the lookup service and uses the lookup service to get references to services offered by other devices. The protocols that are involved are described in the following sections.

2.2.1 Discovery

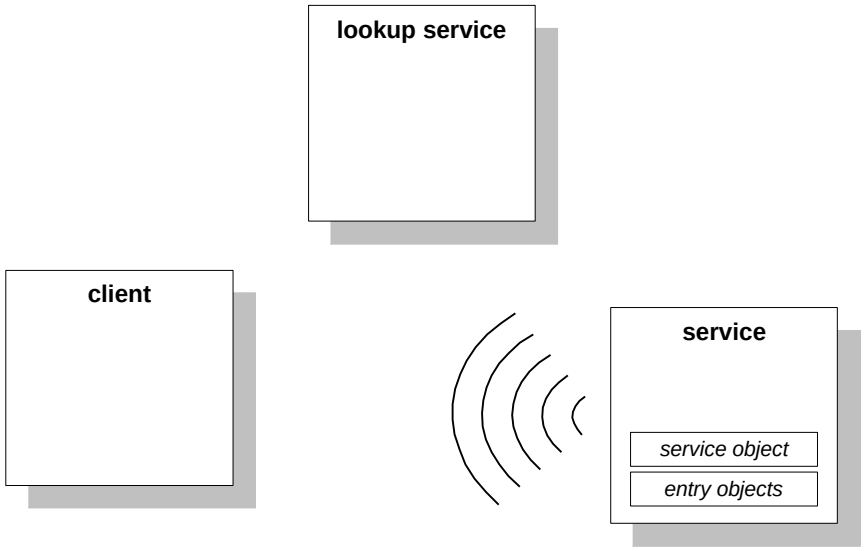


Figure 2.1: *Discovery*. The service seeks a lookup service

When an entity wishes to participate in a Jini system, it first has to obtain a reference to one or more lookup services. To do this the entity has to follow the *discovery* protocol (figure 2.1). Every lookup service belongs to one or more *groups*. When an entity tries to discover a lookup service, it can specify a set of groups it is interested in. This way, only the lookup services belonging to one of the specified groups will respond.

Discovery can be done in three ways:

Multicast Request Protocol: This way, all lookup services on a local subnet can be found. The discovering entity repeatedly performs a multicast by sending an UDP-datagram. This datagram contains the set of groups the entity is interested in and the network address and port of the entity itself. Every lookup service that receives the datagram and is a member of one of the specified groups responds by creating a TCP connection to the discovering entity. The lookup service then sends a proxy back to the discovering entity. This proxy serves as a front-end to the lookup service, and gives the discovering entity access to the lookup service's methods.

Unicast Discovery: This way, a specific lookup service anywhere on the network can be found. This requires the discovering entity to have a LookupLocator for the lookup service, which contains the IP address and the TCP port of the lookup service. The discovering entity creates a TCP connection to the lookup service and sends a request. The lookup service responds by sending a proxy.

Multicast Announcement Protocol: This is used by the lookup service to announce its existence. The lookup service sends a multicast UDP-datagram containing its ServiceID, a LookupLocator and a set of groups. The ServiceID can be used by entities to check if they already know that lookup service. If an entity doesn't know the lookup service and is interested in one of the groups, it can use the LookupLocator to perform Unicast Discovery.

2.2.2 Join

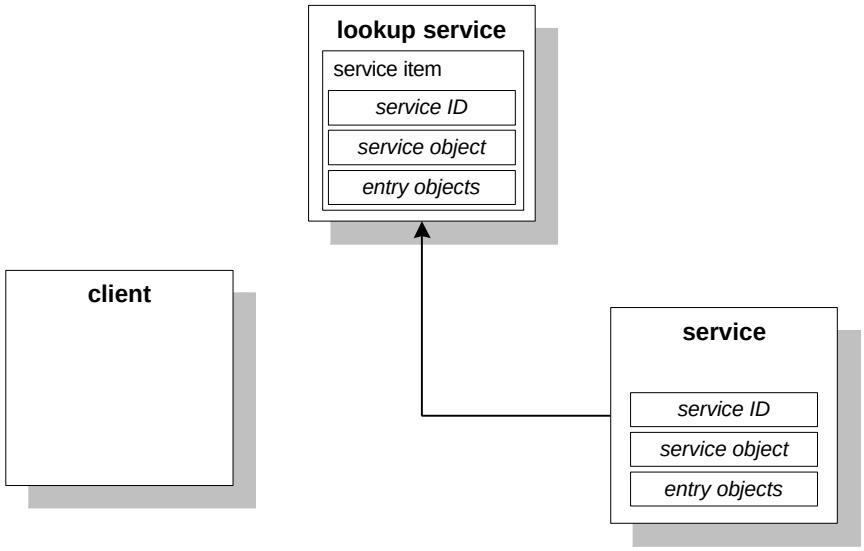


Figure 2.2: *Join.* The service registers itself with the lookup service.

The *join* protocol (figure 2.2) describes the process of registering a service with a lookup service. The registering service provides the lookup service with information about itself. A service is registered by using the lookup service's register method. The arguments to this method are the requested *lease* duration and a ServiceItem. The lookup service returns a ServiceRegistration object. After this the service can be found by other entities.

A lease is a time period during which the grantor of the lease ensures (to its best abilities) that the holder of the lease will have access to some resource. In this case the grantor is the lookup service, the holder is the registered service and the resource is the registration. This means that the lookup service will maintain the registration of the service for the requested time period. When the lease expires, the registration is removed from the lookup service.
Leases can be renewed or cancelled by the holder. The grantor decides whether it grants the requested lease duration, but it can also decide to grant a shorter time period. A lease is represented by a Lease object. The Lease object can be used to renew or cancel the lease or to obtain the lease duration.

A ServiceItem contains a *service object* (a reference to the service), a service's ServiceID and a set of Entry objects. A ServiceID is a unique number that identifies the service. When the service registers for the first time it receives a ServiceID from the lookup service. After that it has to provide this ServiceID with every following registration.
Entry objects contain information about a service: the service attributes. These service attributes can be viewed by other services, so they can find an appropriate service for a certain task.
A ServiceRegistration object contains a Lease and a ServiceID. The service attributes of a service can also be modified through the ServiceRegistration object.

2.2.3 Lookup

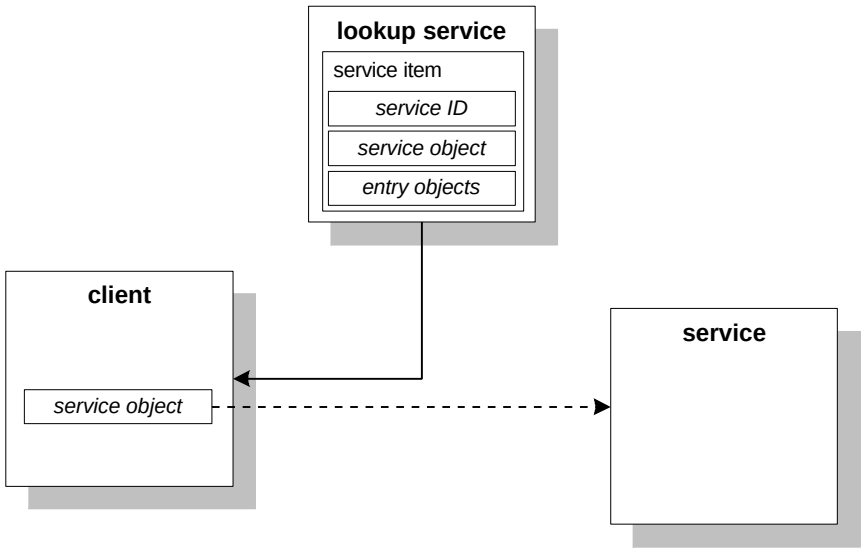


Figure 2.3: ***Lookup.*** The client gets a proxy to the service

The process of finding a service is known as *lookup* (figure 2.3). Entities that want to find a particular service can use the lookup service's lookup method. This method takes a ServiceTemplate as an argument that specifies the services the entity is interested in. A ServiceTemplate can contain a ServiceID, a set of service types and a set of Entry objects. The lookup service returns the ServiceItem objects of all the services that match the service template. This ServiceItem contains the ServiceID of the service and a *service object*. The service object acts as a proxy to the service. The entity can use this service object to communicate directly with the service.

Entities that want to be kept up-to-date about new services or services changing their attributes, can use the lookup service's notify method, which also takes a ServiceTemplate as an argument. When a new service registers that matches the ServiceTemplate or when a change occurs that the entity showed interest in, the lookup service fires an event.

2.2.4 Jini Helper Utilities and Services

The Jini API (Application Programmer Interface) provides a collection of utilities and services that can save the programmer a lot of work.
A helper service is a Jini service that can be registered with any number of lookup services and whose methods can execute on remote hosts. In general a helper service should be of use to more than one type of entity.
Helper utilities are programming components that can be used during construction of Jini services and/or clients. Helper utilities are not remote and do not register with a lookup service. They are instantiated locally by entities wishing to employ them.
There are services and utilities that help with the discovery process, the join process, service discovery, lease renewal and event handling.

2.3 Java Remote Method Invocation (RMI)

So far we discussed how devices can exchange proxy objects by means of the Jini protocols. This section describes how the devices communicate. All communication is done with remote procedure calls. In Java, a remote procedure call is known as a remote method invocation.

A device communicates with a service by calling a method of the service's proxy object. The proxy object serves as a local interface to the remote service. It sends the method call to the service and gives the return from that call to the device that called the method. This mechanism is called RMI, Remote Method Invocation.

What actually happens is the following. The proxy object creates a stream socket connection to the remote object. The protocol that is used for this connection is TCP (Transmission Control Protocol). The stream to the remote object starts with a header that identifies the object and the method that is called. The rest of the stream contains the objects that are passed as parameters to the method. These parameter objects are formatted into a transmittable form by means of the Java Serialization Protocol. The return stream contains an acknowledgement and possibly return objects that are also serialized.

2.4 Framework implementation with NID

This section describes how the framework application that was introduced in section 1.2 would be implemented with NID. It shows how the Jini and RMI protocols are used in a practical application.

The NID-device starts with using Jini's discovery and join protocols to integrate into the system. The device gets references to other devices by performing a lookup at the lookup service. The lookup call contains a template with a number of criteria for the devices it wants to receive data from. The lookup service returns a reference to every device that matches the

criteria defined in the template. These references are proxy objects that allow the device to communicate with other devices by means of RMI calls.
The device uses the references to register itself at the other devices. A device registers itself by giving another device a *listener* object. The other device uses the listener object to communicate. It passes data to the listener object, and the listener object forwards this data to the registered device.
Within the application framework there is a separate program that monitors the devices. This program also performs the discovery protocol to find a lookup service, and performs a lookup to get references to devices. Just as the devices, it uses listener objects to receive data from devices. The monitoring program can also download a user-interface from a device. This allows the operator of the system to interact with a device.

2.5 Conclusion

The NID concept is basically a collection of protocols that allows devices to communicate with each other. First contact between devices is established via the Jini protocols. Devices use Jini's lookup service to exchange references to each other. When a device has a reference to another device, it can send data to the device via RMI calls.
The next chapter will describe Splice, the other system in our comparison.

Chapter 3
Splice

This chapter describes Splice, an architecture developed for large-scale embedded systems. Splice uses a shared data space to distribute data across the network.

3.1 Introduction

The development of Splice began in the early 1980's at Hollandse Signaalapparaten. They needed a software architecture that could be used for large-scale embedded systems like traffic management, process control and command-and-control systems. The complexity of such systems makes them very difficult to implement and even more difficult to modify. A modular approach to the design is therefore necessary. This leads to better designs, less errors and reduced development time.
Traditional modular design methods separate the various functions that a system has to perform into different modules. Today's large-scale embedded systems are often distributed over a large number of interconnected computers. This introduces design problems in the area of communication, distribution of processing and adaptability and extendibility. Functional decomposition only is not adequate to solve these problems.
Splice uses an extension of the traditional approach. It separates the computation part of a system from the coordination part. The computation model expresses the basic tasks that have to be performed by a system. The coordination model expresses the system in terms of processes and the communication between them. This approach improves the modularity of the complete system.

3.2 Software architecture

The Splice software architecture basically consists of two types of components: *applications* and a *shared data space*. The applications are concurrently executing processes, which together perform the functions of the system. They interact only through the shared data space. In this sense Splice bears strong resemblance to Linda [4]. Linda is a coordination language for parallel and distributed processing. It provides a communication mechanism based on a logically shared memory space. This shared memory is called tuple space. Linda provides a library with a small set of operations that may be used to place tuples into tuple space and to retrieve them from tuple space.

In Splice the shared data space is organized as a collection of relational databases. A new database is created by a *sort* definition. The definition of a sort specifies the name of the sort and the names and types of the *fields*. The field types can be integer, double, string and also more complex types like arrays and nested records. Just as with a relational database, one or more fields can be declared *key* fields.
Each data element in the shared data space is uniquely determined by its sort and the value of its key fields. The different sorts enable the application processes to distinguish between different kinds of information.

sort name	object_data (volatile)		
field names	object_id (key)	location	color
field types	long	double[3]	int[3]

sort name	camera_info (persistent)		
field names	camera_id (key)	location	orientation
field types	long	double[3]	double[2]

Figure 3.1 Example sort definitions. The sorts shown in this figure are examples of sorts that could be used in our framework application.

The Splice architecture offers three primitives for interaction with the shared data space. With these primitives an application can store data elements in the data space and retrieve data elements from the data space. The three primitives are defined as follows:

- write(*a*, *x*): inserts element *x* of sort *a* into the shared data space. If the data space already contains an element of the same sort and with the same key value, the old element is overwritten and replaced by the new element.
- read(*a*, *q*, *t*): reads an element of sort *a* that satisfies query *q* from the shared data space. If the data space does not contain an element satisfying the query, the operation blocks until either one becomes available or until the timeout *t* has expired.
- get(*a*, *q*, *t*): operates exactly the same way as the read operation. The only difference is that once an element is retrieved by the get operation, it is hidden from the applications view and cannot be read a second time.

To ensure the consistency of the shared data space the architecture imposes the design constraint that for each sort at most one application shall write data elements with the same key value.

3.3 Distributing the shared data space

To share data amongst application processes running on different computers, the shared data space has to be distributed across a network of computers. The distribution of data is handled by *heralds*.

Each application process has exactly one herald. The herald acts as an intermediary between the application and the shared data space. An application interacts only with its own herald and does this by means of the primitives discussed in the previous section. The herald communicates with other heralds to create a distributed shared data space. Each herald embodies a local database. This local database should contain all data elements of every sort that the application is interested in. This is realized by a subscription protocol.

When an application issues a *read* request for a given sort, the herald checks whether this is the first request for that particular sort. If it is, the herald asks the other heralds for data elements of that sort. It does this by broadcasting a *subscribe* message to all the other heralds. This message contains the name of the sort and the address of the herald.
Every herald that receives the request registers the requesting herald as a subscriber to the requested sort. Then they check if their own application has written any data elements of that sort. If this is the case they send copies of all the written data elements to the subscriber. And

from that moment on they immediately send a copy to the subscriber when their application writes a new data element.
Note that it is important that a herald only sends the data elements that were actually written by its own application. Let's consider what would happen if a herald would simply send all the data elements of the given sort that are in its local database. If the herald has already sent data elements of that sort to another subscriber, the other subscriber will send the same data elements to the new subscriber. This means that the new subscriber will receive multiple copies of the same data element. By limiting the sending of data elements to copies that are written by a herald's own application, a subscriber will receive each copy only once.
Also note that even if a herald doesn't have any data elements of the requested sort in its local database at the moment of the request, it still registers the subscriber. The herald does this because its application may start producing elements of that sort at a later time.

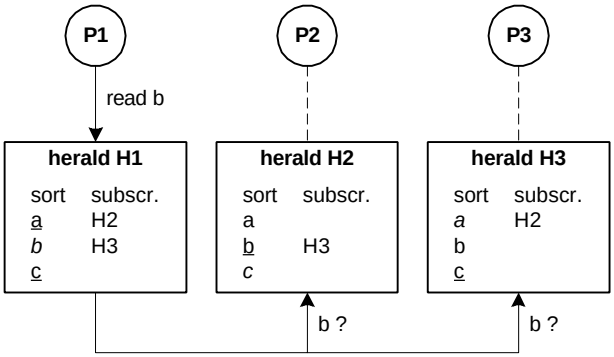


Figure 3.2a. Read request. This figure shows an example Splice system. The *italic* sorts are not in the local database. The plain sorts are in the local database, but there are no data elements present that were written by the herald's own process. The underlined sorts are present in the local database and there are data elements that were written by the local process. Process P1 performs a read request for sort b, but sort b is not in herald H1's database. Therefore, H1 broadcasts a request for sort b.

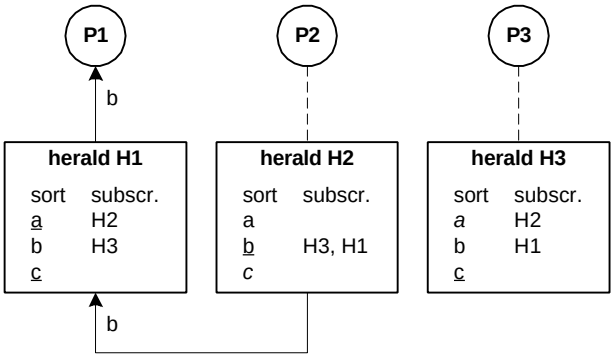


Figure 3.2b. Read reply. Herald H3 does have data elements of sort b in it's local database, but they were not written by process P3, so it does not send any data elements. However, it does register H1 as a subscriber to sort b. Herald H2 has data elements of sort b that were written by it's own process, so it sends these data elements to H1. H2 also registers H1 as a subscriber to sort b. H1 stores the data elements it receives in the local database and responds to P1's read request.

When an application issues a *write* operation, the herald stores the data element in its local database. If this was the first write operation for the written sort, the herald announces this by broadcasting a *publish* message to the other heralds. If this herald did not exist when one or more of the other heralds subscribed to the written sort, it doesn't have their subscriptions. The publish message notifies the heralds of a new publisher, and allows them to add a subscription to the new herald.

After this, the herald of the writing application checks if it has registered any subscribers for the sort that is written. If this is the case, it sends a copy of the data element to every subscriber.
A *get* operation is handled the same way as a read operation. The only difference is that after a data element is passed to the application, it is removed from the local database. We know that a herald receives every data element only once. This ensures that the application will not be able to read the same data element again.

As a result of this protocol, all the data that an application needs will become available in the local database. And each local database only contains the sorts that an application actually reads or writes. Heralds need no prior information about the application they serve, so all heralds can be identical. The communication needs are derived dynamically from the read and write operations that are performed by the application processes. Communication does not have to be considered when designing the application processes. The complexity of the system is thus effectively decreased by this separation between computation and communication. The significant amount of code needed for the communication part is now handled by the standardized heralds.
Note that the use of heralds is a distinct Splice innovation. Other coordination languages suchs as Linda do not have heralds. How data is distributed across a network is not explained. Linda is only concerned with how processes interact with the tuple space, and how they interact with each other through tuple space.

3.4 Volatile and persistent data

The way the protocol is described so far is to store *persistent data*. In Splice, this means that even if an application did not exist at the time that the data was written, it can still read the data. This has some undesirable consequences. Consider the following situation.
A new application enters the system and wants to read data of a certain sort. The application's herald broadcasts the name of the sort. Now, every other herald that already has an application producing data elements of that sort starts sending. Because a lot of heralds start sending at the same time and to the same herald, the network can become clogged. This can interfere with other communications. So, it can take quite a long time before the new application is fully integrated into the system.
To solve this problem, Splice supports *volatile data*. Volatile data is only visible to those applications that are present at the moment the data is written. The distinction between volatile and persistent data is made by adding an attribute to the sort definition that indicates whether the sort is volatile or persistent. The protocol for handling volatile data is almost the same as for handling persistent data. The only difference is in the handling of a write operation. When a data element of a volatile sort is written, the herald does not store the data element in its local database, but sends it immediately to all subscribers. The result of this change is that data elements are only sent at the time they are written, and never afterwards.

In figure 3.1 we can see that the object_data sort is defined as volatile and that the camera_info sort is defined as persistent. Information about objects changes dynamically. The location of moving objects, for example, changes constantly. For this reason, old object

information is not very useful.[1] Therefore, object information is a good example of volatile data. Sensor data should in general be declared volatile, because only recent samples are of interest and new samples invalidate old samples.
On the contrary, information about a camera (e.g. its location, field of view and optical characteristics) usually doesn't change. It is only invalidated when the camera is moved. If a new camera is added to the system, it needs information about the other cameras. This requires that camera_info should be declared as a persistent sort. In general, system configuration data should be persistent.

3.5 Framework implementation with Splice

This section describes how the framework application that was introduced in section 1.2 would be implemented with Splice. Every device has a herald and at least one application process. The herald actually consists of a number of processes. The most important one is the Splice daemon, which takes care of all communication. The tasks of the other processes include managing the global clock and handling persistent data.

For the framework application we need three different kinds of sorts. One sort for publishing information about the camera, one sort for sending object data to other devices, and one sort for user interaction.
The Splice daemon of every device automatically establishes contact with other Splice daemons. Each device starts with publishing information about the camera. Then it reads all the camera information from the other devices. From this information it selects the devices it wants to receive data from, and subscribes to the object data sort of every selected device. Data is sent to other devices by writing it to the shared data space. The Splice daemon then forwards the data to the subscribed devices.
The program that is used to monitor the system also has a Splice daemon. The monitoring program subscribes to the camera information sort, so the operator can see where the cameras are. It can subscribe to the object data sorts of the devices to receive object data. Interaction with the devices is also via the shared data space. Remote commands and replies to those commands are embedded in data elements of the user interaction sort.

3.6 Conclusion

Splice is a distributed shared data space system. The distribution of the data is handled by a number of standard processes that are called heralds. The heralds ensure that all the data that a device needs is available in the device's local database. This is done by means of a subscription algorithm. Communication between devices is only possible through the shared data space.
In the next chapter we will compare NID and Splice for a number of issues that are important for the design of our framework application.

[1] For object tracking, previous locations are necessary. But an object is only tracked by the camera that has the object in its field of view. The other cameras don't track the object, so they don't need its previous locations. Therefore, the previous locations don't have to be kept in the shared data space.

Chapter 4
Crucial issues in device to device communication

This chapter describes the important issues when creating a network of communicating devices. We want to decide which of the two systems that we are comparing (NID and Splice) is best suited for communicating object information in a network of cameras. Therefore, we described a framework for such an application in section 1.2. In sections 2.4 and 3.4 we described how this framework would be implemented with NID and Splice. These implementations will form the basis for our comparison.

4.1 Introduction

The most important issues for our framework application are: security, data representation, data persistency, data consistency, robustness, synchronization, adjustability/flexibility, fault-tolerance, communication protocols, real-time performance, message compactness and scalability. How NID and Splice handle these issues is discussed in the following sections.
In chapter 5 the performance related issues are discussed further on the basis of performed experiments. These issues are: real-time performance, communication protocols, message compactness and scalability.

We will also assign parameters to a number of issues. Each parameter expresses the efficiency of a certain aspect of the system. These parameters are intended for future use for describing distributed networks issues.

4.2 Crucial issues

Each issue will be handled separately in the sections below. We first explain why it is an important issue for the framework application. Then, we describe for NID and Splice how each of them handles that particular issue. Finally, we compare both approaches and decide which approach is the best.

4.2.1 Security

In every networked system security is important. The system has to be protected against intruders that want to read data or interfere with the system in some way. This can be done by encrypting the messages between the devices. For example, IDEA (the International Data Encryption Algorithm) [8] is a well-suited encryption algorithm for this purpose. IDEA is a block-ciphering algorithm, which is considered virtually unbreakable and is also relatively fast. It uses a 128-bit key, which has to be known at the sender and the receiver of the message. The same key is used for encryption as well as decryption.
Typically, two levels of protection are distinguished. The first level protects the system from unauthorized entities that want to join the system. This ensures that no entity can interfere with the system. An entity that joins the system could, for example, send incorrect data to the other devices. The second level protects the data in the system from unauthorized entities that want to read it. This could happen if entities intercept messages that are sent between the devices.

NID

Because a NID system is programmed in Java, it benefits from Java's built-in security. Java is a so-called type-safe language. Type-safe languages ensure that references to objects cannot be forged (i.e. created by an unauthorized entity), and that objects can only be used by holding a valid reference to them. The Java Virtual Machine enforces the Java language's type safety, preventing programs from accessing memory or calling methods without authorization. The Java Classloader ensures that system classes cannot be replaced by user classes.

The first level of protection can be ensured by encrypting the messages associated with the discovery process. The lookup service should only respond to discovery messages that are encrypted with the correct key. The response, which contains a reference to the lookup service, should also be encrypted. The Java Virtual Machine ensures that references can't be forged. In this way, only entities that have the correct key can make use of the lookup service.

If the intruding entity cannot read messages that are not addressed to it, the system is completely protected. If this is not the case, other messages also have to be encrypted. The lookup service's response to a lookup message has to be protected, because it contains references to the devices. The messages between the devices also have to be encrypted so the intruder can't read the data that is in these messages.

Splice

In the Splice system no security is built-in. But the programming model makes it easy to implement security of the first level. The Splice daemons use some sort of discovery protocol to find each other. If the messages of this protocol are encrypted, no unauthorized entity can join the system. Encryption of the subscribe messages is not an option, because unauthorized entities could just send incorrect data to every other device, regardless of the subscriptions.

Just as a NID system, a Splice system is also completely protected by enforcing the first level of protection if intruding entities cannot read messages that are not addressed to it. When an intruder can read messages that are not addressed to it, every message between two devices has to be encrypted.

Comparison

Because the Jini Specification can be simply downloaded from Sun's website, anyone can find out how a lookup service can be accessed. Splice is a commercial product that is not freely available. The protocols that the Splice daemons use are therefore not open to the public. So, it is not very easy to create your own Splice daemon and let it join a Splice system. In that respect, Splice is safer. Beyond that, the security measures that have to be taken are the same for both systems.

4.2.2 Communication protocols

There are two different forms of communication in the framework application. First, the devices have to perform some sort of communication protocol to connect to and integrate into the system. Second, when this is done, the devices need a protocol for sharing data about the detected objects. Each protocol has its requirements, limitations and possibilities.

For good scalability, each protocol should require as few messages as possible and should be implemented efficiently (i.e. require a small number of operations). The efficiency of protocol *P* depends on the efficiency of the implementation *E* and the number of messages *N*. This can be denoted as $P(E, N)$.

NID

The NID system uses Java's Jini Technology for connecting the devices to each other. Every device with a TCP/IP connection to the rest of the network can join the system by means of the Jini protocols. A device can use broadcast messages to discover a lookup service. If the network does not support broadcast or if the lookup service is outside the broadcast domain, the device needs the lookup service's IP-address to connect to it. The devices use the lookup service to exchange references. The references will be used by the devices to send data to each other.

Data is sent by means of Java RMI (Remote Method Invocation) calls. An RMI call creates a TCP (Transmission Control Protocol) connection between the calling object and the receiving object. TCP is a reliable protocol, which means that packets always reach their destination in the correct order.

There are two ways of transferring data. Data can be *pulled* by the receiving object, or *pushed* by the sending object. In our framework application, data is sent as soon as it is available. This means that it is pushed by the device that generates the data. It requires that the devices that want to receive the data have to register at the sending device.

An RMI call is a synchronous data transfer. This means that a method call only returns when both the sending and the receiving object have performed all operations. The advantage of this is that the sending object knows that the receiving object has received the data when the method call has returned. The disadvantage is that the sending object is blocked during the entire transfer. This problem can be solved by calling the method in a separate thread. We will see the drawback of that solution in chapter 5.

Splice

Connecting to the Splice system is done by starting a Splice daemon. The Splice daemon automatically searches the network for other Splice daemons. If the network does not support broadcast messages, the Splice daemon needs the IP-address of one or more other Splice daemons. Once two Splice daemons have contacted each other, they exchange the addresses of other Splice daemons that they have had contact with before.

Data is shared between devices by means of the shared data space. A device can write information to the shared data space and read information from it. It can control which information it receives by means of a subscription paradigm. If it wants to receive data from a certain device, it subscribes to the device's object data sort. The Splice daemons take care of transferring the data from the producing devices to the subscribed devices. The subscription paradigm implies that only push communication is possible.

The data transfer in Splice is asynchronous. When a device writes data to the shared data space, the data is first stored in a local database. So when the write method returns, the device only knows that the data is stored locally. When new data is written to the local database, the Splice daemon tries to transfer it to the local databases of the subscribers as soon as possible. However, the producer of the data will never get an acknowledgement.

The Splice daemons have two different network protocols for transmitting data. The declaration of the data's sort determines whether they use a reliable or an unreliable protocol. The reliable protocol has guaranteed delivery, but it is slower. A big disadvantage of the unreliable protocol is that a lot of packets get lost if the network load is high. This introduces unwanted non-determinism to the system.

Comparison
The communication protocol of NID involves a central marketplace for exchanging references: the lookup service. The disadvantage of this is that network traffic is concentrated on the node of the lookup service. Splice doesn't have this disadvantage. The management of the connected devices is distributed over all nodes.
Furthermore, the asynchronous data transfer of Splice is more efficient than the synchronous protocol of NID. NID, on the other hand, offers more flexibility in the sense that it supports both push and pull communication. However, from a performance point of view, Splice should beat NID. Whether this is also the case in practice, we will see in the next chapter.

4.2.3 Message compactness
Message compactness relates to how the messages between the devices are formatted and sized. In addition to the content, each message has a header, which identifies the recipient and may contain a timestamp and the type of the message.
A scalable system requires that the messages are formatted efficiently and that the size of the messages is as small as possible. The compactness of a message *M* depends on the content *C* and the overhead *H*. This can be denoted as *M (C, H)*.

NID
In the NID system all communication is done by means of remote procedure calls that contain the data that is to be sent. The protocol for the remote procedure calls is Java RMI. The data is represented by Java objects, which are formatted into a byte stream by means of Java's Object Serialization Mechanism. Both the RMI protocol and the serialization of the objects incur some overhead.
The structure of an RMI call is as follows: header (28 bytes) – method identifier (38 bytes) – message data (variable) – acknowledgement (4 bytes) – return data (variable). The overhead of the RMI call is: 28 + 38 + 4 = 70 bytes.
We will calculate the serialization overhead for two example messages. The messages contain locations of detected objects. The location of each object is represented by 3 doubles (a double is 8 bytes in Java). The first message contains 10 locations; the second message contains 100 locations. The efficiency of the serialization depends on how the locations are represented. We discovered that the most efficient representation is to store each location in a Java object and store these objects in an array. This results in the values shown in table 4.1.

locations	data size	serialized	RMI	message size	overhead
10	3·8·10 = 240	371	70	371 + 70 = 441	(441-240)/441 = 30 %
100	3·8·100 = 2400	3071	70	3071 + 70 = 3141	(3141-2400)/3141 = 24 %

Table 4.1. Message compactness of NID.

Splice
In Splice, all communication is handled by the Splice daemons. How the Splice daemons format their messages could not be disclosed. Therefore, we don't know the message compactness.

Comparison
Because we don't know how messages in Splice are formatted, we can't compare the message compactness of the two systems. The data transfer experiment in the next chapter should clarify the situation.

4.2.4 Real-time performance
Our framework application is an example of a so-called soft real-time system [7]. This means that if a result is not produced before a certain deadline, this does not have catastrophic consequences. The performance of a real-time system should be predictable, even at high load.

NID
The NID system is programmed in Java, which is not well known for its real-time performance. In general is an application programmed in Java about 10 times slower than the same application programmed in C.
An advantage of the Java programming language is that it offers the programmer the notion of threads. Threads are lightweight processes, which can be assigned a priority value by the programmer. By performing time-critical operations in a thread with high priority, real-time performance can be enhanced.

Splice
Splice was specifically designed for large real-time systems. Because Splice is programmed in C and uses asynchronous communication, it should inherently be faster than NID. Furthermore, according to [2] the upper bounds of Splice are acceptable for distributed applications where timing requirements are in the order of milliseconds.

Comparison
Because of the difference in programming language and communication protocol, Splice should have substantial better real-time performance than NID. The experiments in the next chapter should prove this.

4.2.5 Data representation
Data representation relates to how data is represented locally. Not all the data that a device generates needs to be sent. A device may want to store some of the data only locally.
There are two kinds of data in this system: data generated by the cameras and data about the cameras. Data generated by a camera will be called *object data*. This is usually the location of the object, and other features of the object that allow it to be identified by other cameras. The data about a camera, which we call *camera information*, is the location and orientation of the camera.
All this data has to be represented efficiently. In this case efficiently means that the data is easily accessible and occupies a small amount of resources. A compact representation of the data is necessary for a scalable system. The compactness of representation R of information I will be denoted as $R(I)$.

NID
All data is represented by Java objects. Object data is represented by the ObjectData class. For every object that a device detects, an instance of this class is created. The attributes of the

ObjectData class contain the coordinates of the location and the values of the other features. The ObjectData objects are stored in a Java data structure, for example a HashMap.
Each device stores its camera information locally and also gives a copy of this information to the lookup service when it registers. To be able to communicate, every device needs references to other devices. These references are represented by *service objects.* A device can ask the lookup service to give it the service objects of devices that match certain criteria. The lookup service uses the camera information to find positive matches. The service objects that are returned by the lookup service are stored in a HashMap, with the device's ServiceID as the key object. A device only stores the service objects of devices it wants to receive data from.
A device also has to store the listener objects of the devices that have registered to receive data. They are stored in a HashMap, with the ServiceID of the registered device as the key object. Service objects and listener objects are stored in a HashMap, so that they can be easily removed when the corresponding device is disconnected from the network.

Splice
In Splice, data is represented by a sort. The most important aspect of a sort is whether it is declared volatile or persistent (see section 3.4). Sorts are represented by C structs. They are stored in the shared data space, which is structured as a collection of relational databases.
Every device needs to have its own sort for publishing object data. Otherwise, data would always be sent to every device in the system. Object data should be declared volatile. This implies that it is not stored in the local database. But it is stored in the database of every subscribing device.
There also has to be one sort that contains information about the camera (e.g. location). This has to be persistent data. All subscribers subscribe to the camera information sort. From this data they select the object data sorts to which they subscribe. Every device has the camera information of every camera in the system in its local database.
The Splice daemon of every device also stores some information. It has to store the network addresses of every other Splice daemon in the system, and it has to store all the subscriptions.

Comparison
The representation of data isn't very different for NID and Splice. A Java object that only stores information is comparable to a C struct. The difference between a HashMap and a relational database isn't very big either. Both store information that can be retrieved by a unique identifier, which is called the key.
The disadvantage of the Splice system is that all camera information is stored locally at every device. This is inherent to Splice's subscription paradigm. Once a device has subscribed to the camera information sort, all available data elements are immediately sent to it. From this we can conclude that Splice requires more resources to store data.

4.2.6 Data persistency
There may be critical data in the system that should never be lost. This data has to be stored somewhere such that it cannot easily be removed. A CD-ROM is an ideal example for persistent data storage, cache memory not at all. Persistency can also be achieved by storing data at multiple locations.
A device needs to know the location and optical characteristics of its camera to calculate the location of the objects it detects. This information about the camera should not get lost, and should therefore be stored persistently.

Object data doesn't have to be stored persistently. The system only keeps track of the current location of an object, so object data is only valid until it is replaced by new object data. When the camera doesn't detect the object any longer, the object data is removed.

NID
Object data is stored in memory by the producer of the data, and by every device that has registered to receive this data. Camera information is stored in memory at the lookup service. If the device has a local disk, it also stores this data persistently on the disk. If this is not possible, because the devices for example only have volatile and read-only memory, we need another solution. We could add a special device that stores the camera information of every camera on a disk. The monitoring station could perform this task.

Splice
Because object data is a volatile sort, it is only stored in the volatile database of the subscribers. This database only exists in memory. Once object data is lost at one node, it is permanently lost there because of the properties of volatile data. When data is read, it is removed from the database. Otherwise the database would keep growing until all available memory is occupied.
Camera information is a persistent sort and is therefore stored in the persistent database of the producer and the consumers. The persistent database is kept in memory and is stored on a local disk. Every device that runs a Splice process has a process that manages the persistent database. It synchronizes the database in memory with the database that is stored on disk. So, when data is lost from memory, it can always be retrieved from disk.

Comparison
Both systems have mechanisms to deal with data persistency. The difference is that in the NID system the programmer has to ensure persistency, whereas the Splice system handles all persistency issues automatically. When a sort is declared persistent, its persistency is ensured by a special Splice process. So, data persistency is more easily achieved with Splice.

4.2.7 Adjustability / flexibility
The system we want to create is not a static system. It has to be adjustable, which means that the system can easily be reconfigured. When a camera is moved, for example, a remote operator should be able to change the camera's field of view.
It also has to be flexible, such that new devices can easily be added and other ones removed. The system should keep on running as usual when a new device is being installed, or when a device is disconnected from the system.

NID
A device can be added to the system at any time. The new device starts with discovering a lookup service and registering at that lookup service. Then it gets a number of references to other devices from the lookup service and registers at those devices. The other devices "see" that a new device has joined the system, so they register at the new device. Now the new device is completely integrated into the system.
A device is removed by canceling the registration at the lookup service. The other devices will notice this, and will discard their reference to it. They also discard the listener object of the disconnected device.
Remote configuration and user interaction with the device can be accomplished by means of the user interface. Every device has a user interface that can be downloaded by any service. In

our case, it would be downloaded by the monitoring program. Actions that are performed on the user interface will be communicated to the device by means of RMI calls.

Splice

A device is added by starting the Splice daemon, which searches the network for other Splice daemons. When the Splice daemon has contacted other Splice daemons, the device can start communicating. It has to start with subscribing to the camera info sort, so it can select the object data sorts of the other devices that it wants to subscribe to. It also has to write its own camera info to the shared data space, so the other devices know there is a new sort to subscribe to.

When a device is removed, it firs overwrites its camera information with a special value, so new devices won't subscribe to its object data sort. Then the device's application process can be terminated. The Splice daemon will notice this, and cancels all subscriptions of the application process. This way, no more data will be sent to the device.

Communication with a Splice process is only possible through the shared data space. This means that messages for interaction have to be embedded in a special sort. It is useful to have one sort for messages from the monitoring program to the devices and another sort for messages in the opposite direction. If we have one sort for both directions, a message from a device to the monitoring program will not only be sent to the user, but also to every device in the system. The devices have no use for such messages, so it is better if they don't receive them. Therefore, we need two separate sorts.

With two sorts, a message from the monitoring program to a device will be sent to every device in the system. Because messages from the monitoring program are usually not very frequent and not very large, this is not a problem. But these messages should have an identifier field that indicates for which device the message is intended. Messages in the opposite direction should also have an identifier field, so the monitoring program knows which device sent the message. Finally, the messages that are used for interaction have to be translated into actions.

Comparison

Flexibility: When a new device is added to a Splice system, it gets the camera information from every device in the system. This can take quite a long time in a system of substantial size. The NID system doesn't have this drawback, because all the device information is stored at the lookup service. In this respect, the NID system is more flexible (assuming that the time it takes to change the configuration is a criterium).

Adjustability: Interaction in the Splice system is not as straightforward as in the NID system. It requires some message coordination, and every message has to be translated into an action. In the NID system, on the other hand, every device has a user interface that allows the user to interact with the device in a natural way. So, we can conclude that the NID system is the more easily adjustable system.

4.2.8 Robustness

Robustness is a measure for the ability of a system to cope with disturbances or even failures within the system. In networked applications, devices may be connected or removed at any time. The system should be able of dealing with these changes without degrading its performance or even crashing.

In a network of devices there are a number of faults that can disturb the system. The most important ones are:

1. *A reference to a device is not valid for some reason, but the device works properly*. The holder of the invalid reference should be able to discover that the device still works, and that the fault is in the reference. It should also be able to restore contact with the device.
2. *A device is suddenly disconnected from the network*. Because the device didn't shut down properly, the system cannot know that the device is no longer connected. Therefore, the references to that device are no longer valid. The system has to deal with this situation and continue operation.
3. *A device does not work, but is still connected*. This can happen, for example, when the device's camera is broken. The device should be capable of detecting that it is not working properly. Since it is still connected to the network, it can notify the system.

NID

The NID system has mechanisms to deal with all three possible faults. Most problems are handled automatically. Every device periodically asks the lookup service for references to the currently registered devices. This way, references to devices that aren't registered any more, are removed from the system. When a device registers at a lookup service, the registration is associated with a lease. The device has to renew this lease periodically. Otherwise, the registration is removed by the lookup service when the lease expires.

1. If a device tries to use an invalid reference, it will get a RemoteException. This means that it can't contact the device for some reason. So, it discards the invalid reference. Invalid references are automatically removed from the system this way. When the device checks the lookup service the next time for a reference to the desired device, it will get a new reference. (The lookup service will certainly have a valid reference to the device.)
2. If a device is disconnected from the network, the lease for its registration at the lookup service will no longer be renewed. So, when the lease expires, the reference to the device is removed from the lookup service. After every device has asked the lookup service for references to the currently registered devices, all references to the disconnected device are removed from the system. Note however that the device was suddenly disconnected. Therefore, there is a period of time that the reference is not valid any longer. This problem is dealt with in the solution to problem 1.
3. The failed device does two things to solve the problem. First, it removes its registration at the lookup service by canceling the lease. This will effectively remove the device from the system over time (see the solution to problem 2). Second, it notifies the monitoring program of the problem, so someone can come and fix it.

The mechanism of periodically checking the lookup service, and periodically renewing the lease ensures that invalid references or references to disconnected devices are automatically removed from the system. However, bad references are not removed from the system immediately.

Splice

Contact between devices is maintained by the Splice daemon of each device. It keeps track of which devices enter the system and which devices disappear from the system. Most robustness problems are handled automatically this way.

1. The Splice daemon solves these sorts of problems automatically. How it does this exactly is classified information.
2. When a Splice daemon tries to contact the disconnected Splice daemon, it will notice that it doesn't get a reply. Consequently, it will remove the subscriptions for the disconnected device. This will prevent it from trying to contact the disconnected device again.
3. When the device detects that the camera is not working properly it overwrites its camera information with a warning. Every device should periodically check the shared data space

for camera information data elements that contain warnings. If a device finds such a data element, it cancels the subscription to the device's object data sort.

How some problems are handled by the Splice system could not be discovered. The articles and documentation about Splice don't reveal all details. The manufacturers of Splice were, for obvious reasons, also unable/unwilling to disclose all details. [2]

Comparison

In the NID system, most problems are solved by means of the leasing mechanism. And if all exceptions that can be thrown are handled properly, the system should keep on running as usual no matter what problem occurs. In Splice, problems are automatically solved by the Splice daemon. It is not clear which approach leads to better robustness. Since for Splice it depends on the implementation of the daemon, NID seems slightly favorable.

4.2.9 Fault-tolerance

Fault-tolerance is related to robustness for the system failure part. A system that is not influenced negatively by failures is fault-tolerant. This means that the system should keep on running as usual no matter how many components fail.

NID

In the NID system, the lookup service is a vital part. If the lookup service fails, new devices can't enter the system, and the problems described in the previous section can't be solved. Fortunately, it is possible to incorporate several lookup services in the system without any problems.

The failure of a device will not degrade the performance of the system. The only problem in that case is that other devices will no longer receive data from the failed device.

Splice

In Splice, the failure of a device will neither cause any problems other than that subscribers to the device's object data sort will no longer receive data.

A device can fail in a number of ways. If the Splice daemon is terminated for some reason, the device is disconnected from the network and all other Splice processes on the same computer are also terminated. A failure of another Splice process can be recovered from by using process replication. There are two ways of doing process replication: passive replication and active replication.

With passive replication one process is executing and the other process is in a sleeping mode. The state of the executing process is periodically stored in the persistent database. When the running process is terminated for some reason, the sleeping process is activated. It first restores the last saved state from the persistent database and continues execution from there.

With active replication two instances of the same process are executing in parallel. When one of the processes is terminated, the other process can take over immediately. This can be useful for time-critical systems.

Comparison

The NID system has the disadvantage that it can't function without a lookup service. Furthermore, the Splice system has mechanisms for process replication. Therefore, we can conclude that Splice shows better fault-tolerance.

[2] Splice is a commercial product. Therefore, it is logical that the manufacturers don't reveal all details. Moreover, Splice is used in military and traffic control systems, so some details can't be disclosed for security reasons.

4.2.10 Synchronization

Ideally, when a message is sent to a number of devices, it arrives instantly at every device. In practice this is not possible. There is always a certain delay before a message reaches its destination. Furthermore, this delay is usually unpredictable. This has two undesired consequences. Messages may arrive in a different order than they are sent, and the sender does not know when the receiver has received the message.

NID

In NID, messages are sent with a synchronous protocol. So, the sender knows that the receiver has received the message when the send operation has ended. Messages can be ordered by giving them a sequence number. This only orders the messages from one sender. Messages from different senders cannot be ordered.

Splice

The Splice system uses an asynchronous protocol for sending messages. Only if the receiver sends an acknowledgement, the sender will know that a message has arrived. For ordering messages, the global clock can be used. Splice has a global clock that gives the same time at every node. The current time can serve as a timestamp for messages. That way, even messages from different senders can be ordered.

Comparison

NID has the advantage of synchronous communication, and Splice has the advantage of a global clock. For our application it is not really important that the sender of a message knows when the message arrives. Therefore, we conclude that Splice has better synchronization properties.

4.2.11 Scalability

Scalability is a measure for how the system behaves when the number of devices is increased. Ideally, the number of devices has no influence on the performance of the system. Practically, when the number of devices is increased this has consequences for the performance and the resource requirements.

The performance is influenced negatively because the number of messages increases. How much the performance is influenced, is determined by the communication protocol $P(E, N)$ and the message compactness $M(C, H)$.

Every device requires more memory because they receive more data and have to store more camera information. The amount of memory that is required is determined by the data representation $R(I)$.

We stated that Splice has a more efficient communication protocol and NID has a more efficient data representation. How the relation is for message compactness, is unknown at this time. However, we can predict that a large NID system requires less memory and that a large Splice system has better performance. In the next chapter we will see if this is the case in practice.

Chapter 5
Experiments

This chapter describes two experiments that were performed to compare the performance of NID and Splice. In the first experiment we measured the performance of the message passing mechanism of both systems. In the second experiment we simulated the communication pattern of our framework application.

5.1 Introduction

So far we have compared NID and Splice on a qualitative level. This chapter will make a comparison on a quantitative level. In other words: we will compare the performance of the systems on the basis of measurements.
Therefore we performed two experiments. In the first experiment we measured a number of data transfers between two processes. This allows us to compare the message compactness and efficiency of the communication protocols of both systems.
In the second experiment we simulated the framework application described in section 1.2. It consists of a number of processes that periodically send messages. We measured how long it takes before a process has exchanged a number of messages with every other process. This way, we can predict the scalability and real-time performance of the two systems.

5.2 Experiment 1: Data transfer

This experiment measured how long it takes to send a number of messages of a certain size from one process to the other. We performed the experiment two times: one time with each process on a different computer, and one time with both processes on the same computer. We want to use the results of this experiment for the analysis of the results of experiment 2. In experiment 2 we will also have more than one process running on the same computer.
The data was transferred from a Sparc5 with 32 MB RAM to a Sparc5 with 64 MB RAM. When we ran both processes on the same computer we used the Sparc5 with 32 MB RAM. We varied the size of the messages. To perform this experiment we had to create a test-program.

5.2.1 The NID program

The NID program consists of two parts: a sender and a receiver. Both processes start with the discovery and join process. The receiver then performs a lookup and gets a reference to the sender. It uses this reference to add a listener to the sender. When the sender has received the listener, the data transfer starts.
Data is transferred by calling the send method of the receiver's listener. The data that is transferred is passed as an argument to this method. The send method is called a number of times. The start-time is recorded just before the first call to the send method. The end-time is recorded directly after the last send call returns. This is a viable approach, because when the send method returns, this means that the receiver has received all of the data.
When the measurement is completed, the measured total time is divided by the number of send calls.

5.2.2 The Splice program

The Splice program also consists of a sender and a receiver. Both processes have a Splice daemon. The Splice daemons automatically establish contact when they are started. If both processes run on the same computer, they share a Splice daemon.

The receiving application process is started first. It subscribes to the sort that will be sent, and then waits for the sender to start sending.

When the sending process is started, it publishes the sort it will write to the shared data space. This is only necessary before the first write operation is performed. After this the measurement starts. The start-time is recorded and it is put in the first data element. Then, a number of data elements are sent by performing subsequent write operations. When the receiver has received the last data element, the end-time is recorded. Because the Splice system has a global clock, the start-time and end-time can be recorded on different computers. Afterwards, the total time is divided by the number of write operations.

We cannot record the end-time at the sender when the last write operation returns. The return of a write operation only means that the local herald has received the data. It does not mean that the data has been transferred to the herald of the receiver, and that the receiver has received the data.

5.2.3 Results

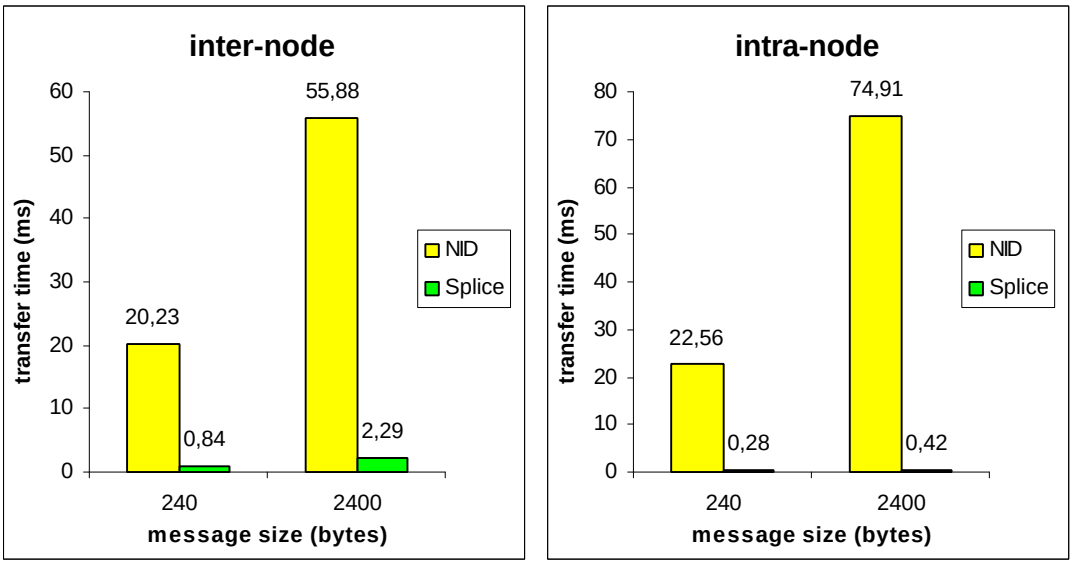
Before we look at the results of this experiment, we will first examine what we have actually measured. When a message is sent from one process to another, a number of actions are performed.

First, the data of the message has to be prepared for transmission at the side of the sending process. This means that data has to be formatted into a transmittable form, and headers have to be created. This can be handled differently by NID and Splice.

After this, the data can be sent across the network. The network and transport layers divide the data into packets and deliver it at the right computer. NID and Splice both use a reliable protocol for this experiment. Splice also supports an unreliable protocol, which is usually faster, but does not guarantee delivery. We decided to let Splice use the reliable protocol, so that they both use the same sort of protocol, since we don’t want to involve network issues in this experiment.

Finally, the receiving process has to remove the headers, and translate the data back from the transmittable form. Then, the data can be stored.

NID versus Splice



(a) **(b)**

Figure 5.1. Data transfer: NID versus Splice. The left graph (a) shows the results of the data transfer experiment with two processes on different computers. The right graph (b) shows the results of the experiment with both processes on the same computer. The data in the messages represents locations of detected objects. A location is represented by 3 doubles, which is 3 * 8 = 24 bytes. So, 240 bytes represent 10 locations.

In figures 5.1.a en 5.1.b we see that Splice is much faster than NID. In the inter-node test Splice is about 25 times faster, and in the intra-node test Splice is almost 200 times faster for the larger message size. For the inter-node test there are two factors that can explain the much better performance of Splice. First, the transmittable form of the data is much smaller for Splice than for NID. This results in a shorter transmission time. Second, the translation of data to a transmittable form takes less time.

The first factor is not very likely to cause a big difference. For NID we know exactly how the data is formatted (see 4.2.3 Message compactness). The overhead of a message with 10 locations (the smaller message size in figure 5.1.a) is 30 %. We don't know the exact overhead of a message in Splice. But even if it is 0 %, a Splice message is 30 % smaller than a NID message, and would only result in a 30 % shorter transmission time.

The second factor is more likely to cause a big difference. First of all, the NID processes are written in Java, and the Splice processes are written in C. In general, a program written in C is about 10 times faster than the same program written in Java. Furthermore, the Object Serialization Mechanism that the NID process uses to format the data costs probably more time than the Splice process' formatting operations. This is because the purpose of the object serialization is that data can be exchanged between two different computing platforms. Splice does not support this, and therefore probably has a simpler mechanism for formatting data. Hence, we can explain that Splice transfers data between two computers about 25 times faster than NID because the formatting operations take much less time.

For the intra-node test we have to take another issue into account. If two Splice processes run on the same computer, they share their Splice daemon. In that situation, the data isn't formatted for transmission. The NID processes, however, "don't know" that they are on the same computer, so they still perform all the operations to format the data into a transmittable form. For this reason, the difference is even larger for the intra-node test.

This section showed a small part of the results that allowed us to compare NID with Splice. Next, we will analyze the entire set of results separately for NID and Splice.

NID

In the previous section we saw that the measurements of the NID program resulted in much longer times than the Splice program. From this we can conclude that, for the NID experiment, the transmission time is only a very small part of the total time needed for the data transfer. The major part of the transfer time is needed for object serialization.

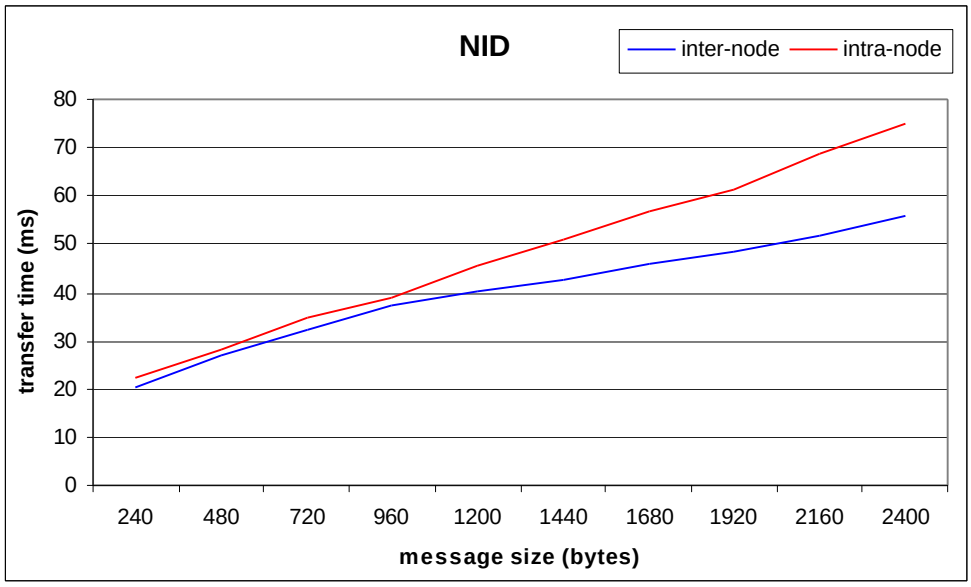


Figure 5.2. Data transfer with NID.

In figure 5.2 the relationship between the message size and the transfer time is shown for both the inter and intra-node communication. It can be seen that both lines are nearly straight. This is logical, because the number of operations needed for object serialization increases linearly with the size of the object that has to be serialized.

There is also a logical reason why the intra-node data transfer takes more time than the inter-node data transfer. Communication between two processes on the same computer does not require network transmission, but we already concluded that the network transmission time is only a very small part of the total time. What is more important is that both processes now use the same processor to perform all operations. This slows down the entire data transfer. Because the number of available processors is divided by two, one might think that the data transfer will take twice as much time. However, there is not much overlap in the activities of the sending and the receiving process. The sending process begins with a number of operations before the data is transmitted, and the receiving process will start its operations not until it has received the first packet of data.

Splice

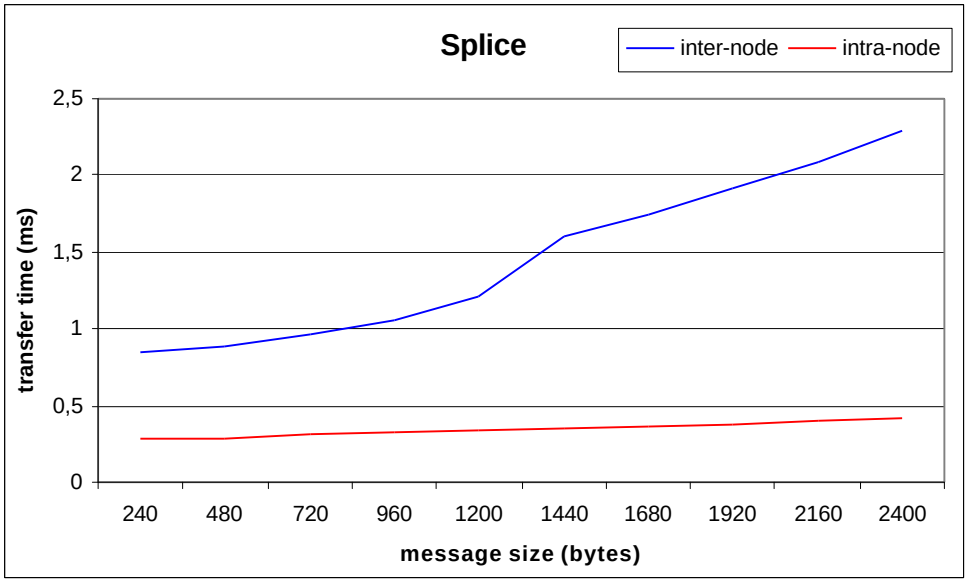


Figure 5.3. Data transfer with Splice.

In figure 5.3 we see that for Splice the line of the intra-node data transfer is almost perfectly straight. This is not very strange. When data is transferred between two Splice processes on the same computer the only thing that happens is that the sending process writes data to shared memory space, and the receiving process reads it from the same memory space. Both operations increase linearly with the number of bytes that is transferred.
The line of the inter-node data transfer is not entirely straight. There is a clear jump between the message sizes of 1200 bytes and 1440 bytes. We can't say that the network transmission time doesn't play a role in the measurement of the Splice program. The measured transfer times are so small that the network transmission time probably does play a role. The jump in the line could therefore very well be a result of a jump in network transmission times. Note that the maximum packet size of an Ethernet packet is 1500 bytes [17]. The message of 1440 bytes combined with some overhead bytes doesn't fit into one packet. So, there are two packets needed to transmit a message of this size. The jump from one packet to two packets explains the jump in the line.

Conclusions

From the jump in the inter-node line in figure 5.3 we can deduce the message compactness of the Splice system. We know that the message of 1200 bytes is the largest message in our experiment that fits into one packet, and that the message of 1440 bytes is the smallest message that needs two packets. This means that the largest message that fits into one packet is between 1200 and 1439 bytes. If it is 1200 bytes, the overhead is: (1500 – 1200) / 1500 = 20 %. If it is 1439 bytes, the overhead is: (1500 – 1439) / 1500 = 4 %. So the actual overhead is between 4 % and 20 %. In section 4.2.3 we calculated that the overhead in the NID system was between 24 % and 30 %. Hence, we can conclude that the message compactness of the Splice system is better than the NID system.
The communication protocol of the NID system involves object serialization. We concluded that this mechanism takes much more time than the mechanism that Splice uses for formatting data. So, Splice has a far more efficient communication protocol. The advantage of the NID system's object serialization is that data can be transferred between different platforms (e.g. from Windows to Unix). This is not possible with Splice.

5.3 Experiment 2: Framework simulation

In this experiment we simulated the communication that will typically take place in the framework application that was described in section 1.2. This means that we had a number of processes on a number of computers that communicated with each other. We let each process periodically send a message to every other process. This is not entirely in accordance with our framework application, which lets devices only communicate with devices nearby. But this does not hinder our comparison, since it effects both systems in the same way.
There was, however, a small problem. We only have a trial license for the Splice system. This license disallows networks larger than 5 nodes. This means that we were unable to use more than 5 computers for our experiment. A typical implementation of our framework application has a lot more than 5 devices. That's why we ran more than one process on a computer for this experiment. This allowed us to measure the performance of a much larger system. This way, we actually simulated a system in which more than one device makes use of the same computing resource.
In practice, the computer will be used for more than just handling communication. It also processes the video sequence from the camera to detect moving objects. This is a very computationally intensive task. In our experiment we assume an ideal situation. This means that video processing takes no time, memory or processor cycles, and that the size of the data that is sent between the processes is negligible.
Of course this is not an accurate representation of a real system. But this experiment does allow us to compare the performance of NID and Splice for a realistic communication pattern. Besides, we are not interested in the video processing part of the system, but only in the communication part.
What we measured is how long it takes before a process has received 50 messages from every other process. Every process sends its data at a rate of 10 Hz. This should be approximately the frame-rate of the object detection module in a real system. On every computer we always had one process that not only sends data, but also did the measurements. We also monitored the memory usage and the processor usage of the processes. We varied the number of computers and the number of processes per computer.
The experiment was performed on a heterogeneous network of computers. We used 4 Sparc5s and one UltraSparc. Two of the Sparc5s have 32 MB RAM, the other two have 64 MB RAM. The UltraSparc has 192 MB RAM.

5.3.1 The NID program

Every process starts with performing the discovery and join protocols to connect to the Jini network. After this the process performs a lookup to get references to the other devices in the system. It adds a listener object to every device that it gets a reference to. Then it records the start-time and starts sending data.
The process calls the send method of every listener object that it has received. There are two ways to do this. Each call to the send method of a listener object is done in a different thread, or every call is done in the same thread. The latter means that the next call can only be done when the previous one has returned. We do the experiment for both ways to see the advantages and disadvantages of each way.
After the data is sent to every registered device, the process sleeps for 100 milliseconds (since we want to transmit data at a rate of 10 Hz). This is repeated indefinitely. When a listener object's send method is called, it calls its device's received method. This method stores the data at the receiving device.
The description so far applies to both the measuring process and the "normal" process. In addition to storing the data, the measuring process also counts the messages it receives with

the received method. When it has counted 50 messages from every device, it records the end-time. After this the counters are reset, and the start-time is recorded again.

5.3.2 The Splice program

There are two different kinds of application processes for this test. One process only reads and writes data, the other process also measures how long that takes. We first describe the process that only reads and writes.
When this process is started it registers itself as a subscriber to the sort that will be sent, and also announces that it will publish this sort. Then, it adds a filter to its local database that will prevent the process from reading data that was written by itself. It creates a data element of the sort that will be sent, and puts its application identifier in it. The application identifier is a unique number for every Splice process. The measuring process can distinguish between data from different processes by reading the application identifier contained in every data element.
After these initialization steps, the process enters a loop. In this loop, the process first reads all available data and then writes its data element to the shared data space. After this, it goes into a sleeping mode for 100 milliseconds. This loop repeats itself indefinitely.
The measuring process performs the same actions as the other process. In addition to this, it counts how many messages it has received from each process. When it has counted 50 messages from every process, it records the end-time. The start-time is recorded just before the counting starts.

5.3.3 Results

From the results of experiment 1 we can already make a prediction about the results of this experiment, which are shown in figure 5.4. Because NID needs a lot more time to transfer data than Splice does, we can predict that NID also takes more time for this experiment. Furthermore, we concluded that NID performs a lot more operations to send a message. This will have a negative effect on the CPU load and will prevent us from running a lot of processes on one computer.

NID versus Splice

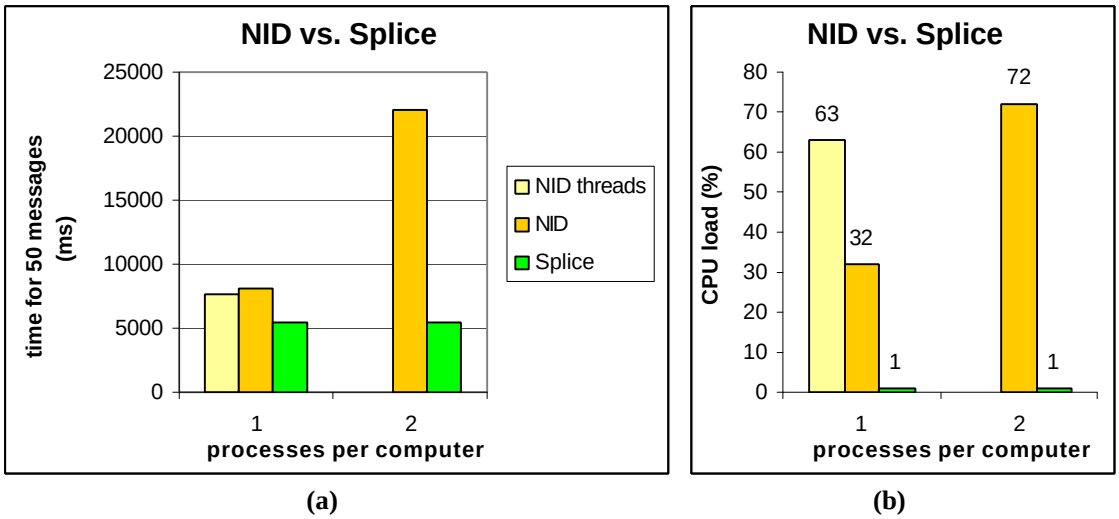


Figure 5.4. Framework simulation: NID vs. Splice. The left graph (a) shows the averages of the measurements on all 5 computers. The right graph (b) shows the averages of the CPU loads of the 4 slowest computers.

The first thing that can be noticed about figure 5.4 is that it only shows the measurements for one process per computer and two processes per computer. The reason for this is that we

weren't able to perform measurements for a NID system with more than two processes per computer (which is a total of 10 processes). The CPU load with three processes per computer was so high that leases weren't renewed and connections between processes timed out. The results of these problems were that references were discarded, and processes were disconnected from the system. For the NID program that sent each message in a different thread, these problems already occurred with two processes per computer.
The NID experiment as well as the Splice experiment showed no big differences between the transfer times measured on different computers. This could be expected, because every process has to wait for the slowest process. The CPU load, on the other hand, was much lower on the fast UltraSparc than it was on the slow Sparc5s. We only look at the CPU loads of the slow computers, because these limit the performance of the entire system.

Let's see if we can predict the measurements of this experiment by calculating the number of messages that are sent, and multiplying this with the transfer times measured in experiment 1. The following formula can be used for this:

$$total\_time = 50 \cdot (100 + processes\_inter \cdot time\_inter + processes\_intra \cdot time\_intra)$$

The formula consists of the following factors:
- 50 : the number of iterations in each measurement;
- 100 : the sleep time in each iteration;
- *processes_inter* : the number of processes on other computers that send a message;
- *time_inter* : the transfer time of a message from a process on a different computer;
- *processes_intra* : the number of processes on the same computer that send a message;
- *time_intra* : the transfer time of a message from a process on the same computer.

If we apply the formula to the experiments that are shown in figure 5.4 we get the values of table 5.1.

processes per computer	predicted values		measured values	
	1	2	1	2
NID	9046	14220	8098	22051
Splice	5168	5350	5494	5498

Table 5.1. Prediction of the measurements. This table shows the predicted and measured times in milliseconds. We used the measured transfer time of the smallest message (240 bytes) for the prediction. The table doesn't show the predictions for the NID program with threads. The reason for this is that the formula doesn't apply to that program, because it sends all messages concurrently.

We see that the prediction for the NID system with one process per computer is somewhat high. This can be explained by the fact that the message size used for this prediction (240 bytes) was a lot bigger than the size of the messages in experiment 2 (8 bytes).[3] There is also a good explanation why the prediction of the NID system with two processes per computer is a lot lower than the measured value. The CPU load was 72 % for this experiment. In every experiment we saw that a CPU load above 70 % results in much longer response times.
The predictions for the Splice system are both a bit lower than the measured values. The CPU load was only 1 % for both experiments, so that can't be the explanation. A possible explanation is that the measured inter-node transfer time doesn't apply to all transfers. The

[3] Even though this is a considerable difference, the prediction is still viable. From the results of experiment 1 we can deduce that the difference in transfer time between 240 bytes and 8 bytes will be approximately 25 % for NID and 10 % for Splice.

data transfer experiment was performed on two computers physically located directly next to each other. For experiment 2 we used the same two computers plus another computer that was in the same room, and two computers that were in a different room on a different floor. The transfer time between two computers on different floors is usually longer than two computers in the same room. This could explain the difference of a couple of hundred milliseconds between the predicted values and the measured values.

NID

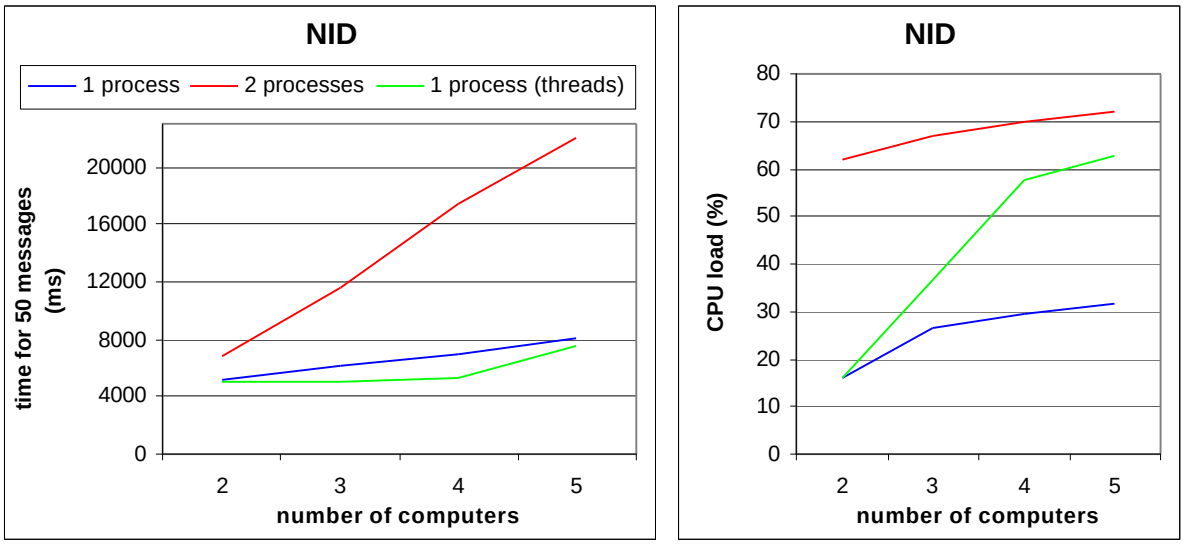


Figure 5.5. Framework simulation with NID. The left graph (a) shows the averages of the measurements on all 5 computers. The right graph (b) shows the averages of the CPU loads of the 4 slowest computers.

Because we weren't able to increase the number of processes per computer very much for the NID experiment, we didn't get much of measurement data. Therefore, we performed the experiment with a smaller number of computers. This gives us some additional measurements and allows us to make a better prediction about the scalability.

In figure 5.5.a we see that the measured time increases linearly with the number of processes. This applies to the experiment with one process per computer as well as the experiment with two processes per computer. The experiment with the program that uses threads, on the other hand, does not increase linearly. It shows a large increase from four to five processes per computer.
The linearity of the two experiments without threads can be explained as follows. Because every process sends data to every other process, the total number of messages m is a quadratic function of the number of processes n:

$$m \equiv n^2 \qquad (1)$$

This means that the number of operations o that has to be performed to send the messages is also a quadratic function of the number of processes:

$$o \equiv m \equiv n^2 \qquad (2)$$

The total computing power of the entire system c is a linear function of the number of computers p and since we have one process per computer we get the following relation:

$$c \equiv p \equiv n \qquad (3)$$

The time t needed to perform the operations has the following relation to the computing power:

$$t \equiv o/c \qquad (4)$$

The result is that the time needed to process the messages is a linear function of the number of processes:

$$\left.\begin{array}{l} o \equiv n^2 \\ c \equiv n \\ t \equiv o/c \end{array}\right\} \Rightarrow \quad t \equiv n^2/n \quad \Rightarrow \quad t \equiv n$$

The reason why the experiment with threads does not show an entirely linear increase, is that the CPU load is a limiting factor. In figure 5.5.b we see that the CPU load increases linearly from two to four processes. From four to five processes the increase declines. The reason for this is that the processors can't handle a higher load. We will see the same thing happening with the Splice experiment. Because the processors can't provide the required amount of operations, the linear increase in time can't be maintained.
We just stated that the limiting factor in the threads experiment was the CPU load. But in figure 5.5.b we can also see that the experiment with two processes per computer showed a higher CPU load than the threads experiment. This CPU load is the combined load of the two processes that are running. Apparently, the processor can handle a higher load when the load is distributed among two processes, compared to one process with a large number of threads.

processes per computer	with 3 computers			with 5 computers		
	192 MB	64 MB	32 MB	192 MB	64 MB	32 MB
1 (with threads)	15	14	13	16	14	13
1	14	13	13	15	14	13
2	15	14	10	15,5	14	10

Table 5.2. Memory usage in NID. This table shows the memory usage per process in megabytes. Separate values are shown for the different computers.

In addition to the CPU load, we also monitored how much RAM memory each process used. The memory usage didn't play an important role in the experiments with only one process per computer. In table 5.2 we see that on a computer with more memory the processes use slightly more memory than they do on a computer with less memory. But overall is the memory usage almost the same on each computer.
In the experiment with two processes per computer, on the other hand, we see that the processes on the computer with 32 MB only use 10 MB. The reason for this is that there simply isn't more memory available. The operating system takes up a certain amount, and what remains is not enough for two processes to function optimally. The result is that the whole system is slowed down by the computers with insufficient memory.

The framework application is intended to have one camera per computer. The experiment with one process per computer applies to that situation. We want to know how much the data transfer time and the CPU load increase with every device that is added to the system. These values can be calculated by calculating the slope of the lines in figure 5.5:
- time increase: (8098 – 5133) / (5 – 2) = 988 ms per device
- CPU load increase: (32 – 16) / (5 – 2) = 5,3 % per device

We can use these values to predict the performance of a larger system under the same circumstances. They can also be used to compare the scalability of NID and Splice.

Splice

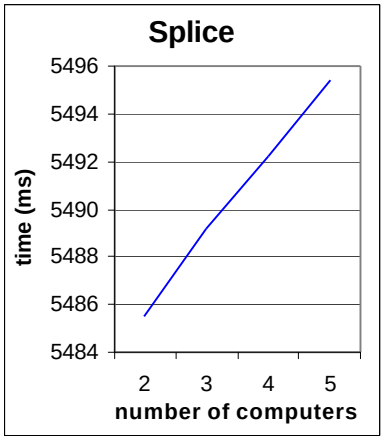
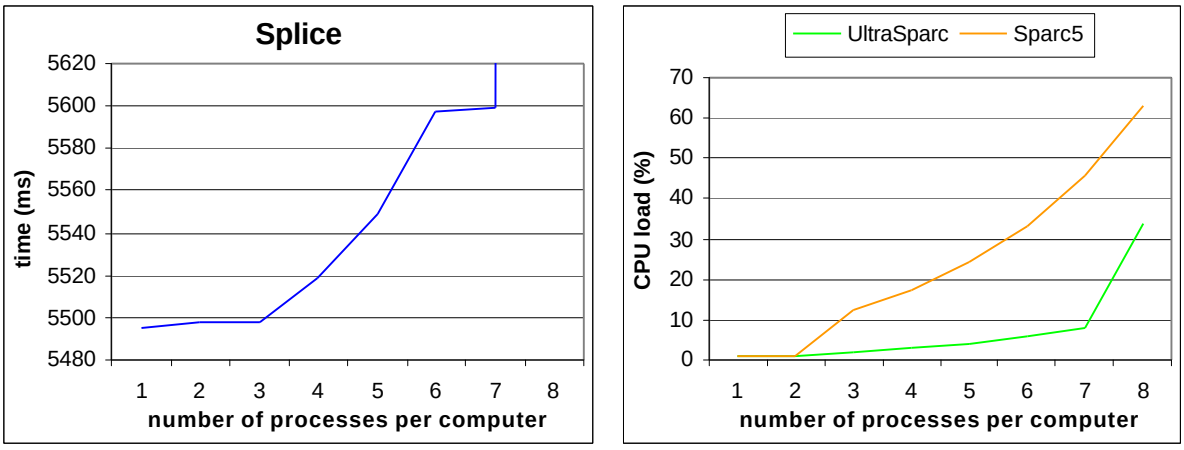


Figure 5.6. Framework simulation with one process per computer. The values are the averages of all participating computers.

The line in figure 5.6 shows that the measured time of the Splice system increases linearly with the number of computers, just as the NID system. This allows us to calculate the time increase the same way as we did for NID:
- time increase: (5495,4 – 5485,5) / (5 – 2) = 3,3 ms per device
The CPU load remained constant at approximately 1 % during the experiment shown in figure 5.6. This makes it impossible to calculate a realistic value for the CPU load increase.

The reason for the linearity of the line in figure 5.6 is the same reason as the linearity of the NID experiment. Why the line of figure 5.7.a looks somewhat strange can be explained by looking at figure 5.7.b.



(a) **(b)**

Figure 5.7. Framework simulation with more than one process per computer. The left graph (a) shows the measured time averaged over all 5 computers. The right graph (b) shows the CPU loads of the different computers. The values for Sparc5 are the averages of the CPU loads of the 4 Sparc5s.

The line for the Sparc5 in figure 5.7.b shows roughly the same behavior as the line of figure 5.7.a. Because the slowest computer determines the performance of the entire system, the CPU load of the Sparc5s predicts the measured time on every computer. That's why the two lines look similar.

The behavior of the lines seems to be quadratic. This can be explained as follows. Equations 1, 2 and 4 on pages 37 and 38 apply to this situation. Because the number of computers is constant in this experiment, the total computing power is also constant. Therefore, we get the following relation in stead of equation 3:

$$c \equiv p \equiv 1$$

The result is that the CPU load is a quadratic function of the number of processes in the system:

$$\left.\begin{array}{l} o \equiv n^2 \\ c \equiv p \equiv 1 \\ t \equiv o/c \end{array}\right\} \Rightarrow \quad t \equiv n^2/1 \quad \Rightarrow \quad t \equiv n^2$$

In figure 5.7.a we see that the line shoots up after 7 processes. This is because the CPU load is so high that the processes can't keep up any more with all the incoming data. This also effects the memory usage as we will see in figure 5.8.

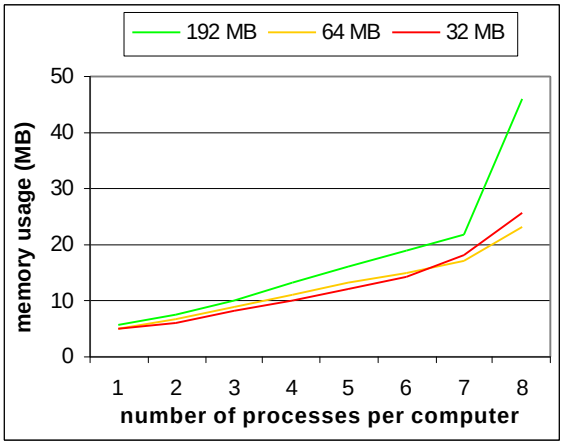


Figure 5.8. Framework simulation with more than one process per computer. The graph shows the memory usage on the different computers.

The lines show the same behavior for all computers. The only difference is that the processes on the computers with more memory also use more memory. We saw the same thing in the NID experiment. We see that the memory usage increases linearly up to 7 processes and than shows a much bigger increase. The linear increase is because every additional process requires a certain amount of memory. The big increase from 7 to 8 processes is because the CPU load becomes so high that the processes are unable to read and subsequently remove all data from the shared data space.

Conclusions

The large number of operations a NID process has to perform to send a message severely limits the scalability of the NID system. We calculated that every additional NID process increases the time by 988 milliseconds and the CPU load by 5 %. In the Splice system these values were respectively 3,3 milliseconds and too small to calculate. The fact that the maximum number of processes for the NID system was 10 and for the Splice system was 35, also reflects the much better scalability of the Splice system.

In every experiment the Splice system showed better times than the NID system, so we can conclude that the real-time performance of the Splice system is better than the NID system.

Chapter 6
Discussion

In section 1.2 we described an application framework for a multi-camera system that may be implemented with either NID or Splice. We introduced the most important issues for this application and compared how NID and Splice handle each of these issues. This resulted in a thorough examination of both systems. The discussed issues and each system's most notable strengths and weaknesses are shown in table 6.1.

Issue	NID	Splice
security	+ Java built-in security – protocols openly available	+ protocols not available
communication protocols	– central lookup service ? synchronous data transfer	? asynchronous data transfer
message compactness	? RMI header, data formatted with object serialization	? unknown
real-time performance	+ threads with priorities – programmed in Java	+ asynchronous data transfer + programmed in C
data representation	? data stored in Java objects	– camera information at every node ? data stored in C structs
data persistency	– ensured by programmer	+ ensured automatically by Splice daemon
adjustability / flexibility	+ easy adjustability with RMI calls + Jini's flexibility	– complicated device adjusting – large data transfer at start-up
robustness	+ lease mechanism + exception handling	+ automatically by Splice daemon
fault-tolerance	– lookup service vulnerable part	+ process replication
synchronization	+ synchronous communication	+ global clock – asynchronous communication
scalability	+ data representation – communication protocols	+ communication protocols – data representation

Table 6.1. Characteristics of NID and Splice. The table shows the characteristics of each system for the issues discussed in chapter 4. Legend: + favorable; ? neutral; – unfavorable.

For our comparison, the most interesting issues in table 6.1 are data persistency, adjustability / flexibility, robustness, real-time performance and scalability. From the table we can immediately conclude that NID is more flexible and easier adjustable than Splice. The Jini technology used by NID is specifically designed to interconnect all sorts of devices, and because each device is accessible through RMI calls, it can be very easily adjusted. Jini's leasing mechanism and Java's exception mechanism make a NID system practically invulnerable to disturbances. This ensures the NID system's superior robustness.
Splice, on the other hand, has a more sophisticated mechanism for dealing with data persistency. The programmer only has to declare whether data is volatile or persistent, and the Splice daemon automatically ensures the data's persistency. The table also suggests that the Splice system has better real-time performance and scalability. This is shown more clearly in table 6.2.

	NID		Splice	
data transfer size	240	2400	240	2400
- inter-node (ms)	20,23	55,88	0,84	2,29
- intra-node (ms)	22,56	74,91	0,28	0,42
nr. of processes	5	10	5	10
- time (ms)	8098	22051	5494	5498
- CPU load (%)	32	72	1	1
time increase (ms)	988		3,3	
CPU load increase (%)	5,3		0	

Table 6.2. Summary of experimental results. This table shows the most important results from the experiments that were performed in chapter 5.

Real-time performance and scalability are probably the most important issues for our framework application. Real-time performance is important because the time it takes to send object information from one camera to another is crucial for object tracking. If that time is too long, it is impossible to track a moving object. Scalability tells us something about the number of cameras that we can connect before the system cannot be used any longer. This can happen if the system gets too slow or if a certain limit in computational power or memory usage has been reached. For a system with many cameras good scalability is essential.
The results of both experiments that were performed in chapter 5 showed very large differences in real-time performance and scalability between NID and Splice. In table 6.2 we can see that an inter-node data transfer with Splice is about 25 times faster than the same data transfer with NID. The CPU load with 10 processes is 72 % with NID and only 1 % with Splice. The measured time is also a lot better in Splice. So, Splice clearly shows better real-time performance.
The increase in time and CPU load per added process shows the largest difference. The NID system requires 988 milliseconds extra for every additional device to send all messages, whereas the Splice system only requires 3,3 milliseconds extra. The increase in CPU load was so small in Splice that it couldn't be measured. This implies that the Splice system is far better scalable than the NID system.
Note that the calculated increase in time and CPU load only applies to a system in which each device communicates with every other device. In practice, every device only communicates with devices that are nearby. This reduces the increase in time and CPU load, and makes the system better scalable.

One of the reasons for the large differences is that we performed the experiments mainly on very slow computers (Sparc5s). If the experiments were performed on faster computers, the differences would probably be smaller. Nevertheless, the experiments would still show a substantial difference. The advantage of using slow computers is that it makes very clear that there is a large difference in performance between the two systems.

We concluded that the main difference in performance between NID and Splice can be attributed to the difference in message formatting. A significant speed-up of the NID system can be achieved if we use another way to send data in-stead of RMI. Then, we can circumvent the object serialization step, which requires a lot of computation. We can use Java's Socket class to create custom sockets that allow us to transmit data in raw byte form. However, this severely complicates all communications and strongly deviates from the NID concept.

Furthermore, according to Boasson object-oriented programming languages are not suitable for real-time distributed embedded systems [3]. In such applications, the relationships between processes is based on the sharing of data, which strongly violates the object-oriented paradigm. According to this observation, NID is inherently unsuitable for distributing data between devices, since the NID concept is based on object manipulation.
The results from our experiments certainly coincide with this conclusion. On the other hand, the comparison in chapter 4 showed that the NID concept has many desirable characteristics for a system of communicating devices.

We simulated a large system by running several processes on the same computer. This had some unwanted consequences in both systems, and influenced the results of the experiments. In experiment 1 we saw that in the NID system an intra-node data transfer took more time than an inter-node data transfer. The Splice system showed the opposite result. So, when we performed experiment 2 with more than one process per computer, this had a different effect on each system. The NID experiment took more time than it would have taken with only one process per computer and with the same total number of processes. In this case, the Splice experiment took less time. Hence, Splice seemed more scalable than it really is, and NID seemed less scalable than it really is in the experiments with several processes per computer.
In a practical application, every device will have its own computer. This reflects the situation with one process per computer. So, the results of the experiments with more than one process per computer are not very useful. To make better predictions about systems with a larger number of processes, we have to redo experiment 2 with a larger number of computers. Especially predictions about the increase in memory usage of both systems, and the increase in CPU load of the Splice system, require a larger experiment.

Although the comparison was done for one particular application, the outcome is useful for the design of distributed embedded applications in general, since the discussed issues are crucial for every system involving communicating devices.

The parameters that were introduced in chapter 4 can also be used for the design of a new application. However, this requires some further exploration of the parameters.
The parameters can be used, for example, to show relationships between certain aspects of the system. It would also be interesting to create a formula from the parameters that predicts certain performance and scalability aspects of the system. The designer examines a number of aspects of the system, so that he or she can assign a value to each of the parameters. By filling in those values in a formula, the designer can then see what the performance of the system is, without having to implement and test the whole system.

Chapter 7
Conclusions

In this study we made a quantitative and qualitative comparison of two concepts for distributing data, namely NID and Splice. The goal of this comparison was to be able to choose one of the two systems for a certain application. The reason for this study is the development of a distributed surveillance system. For this application we need a system that facilitates communication between devices. We identified two candidates: the NID concept and Splice.

In section 1.2 we described a framework application. We first identified the most important issues for the design of such an application. Then, we compared NID and Splice for each issue. We concluded that NID shows better flexibility, adjustability and that it guarantees slightly better robustness. Splice has better characteristics in the areas of synchronization, data persistency and fault-tolerance. The issue of security shows no apparent winner.
Chapter 5 shows the results of two experiments that we created to compare the performance of both systems. The first experiment showed that Splice's communication protocol is more efficient than that of NID, and that the message compactness of Splice is also better. Overall was a data transfer with Splice 25 times faster than with NID. The second experiment showed that the real-time performance and the scalability of the performance and CPU load was also a lot better with Splice. The main reason for the difference in performance between NID and Splice is the NID system's computationally intensive communication protocol.

The outcome of this study could already be predicted by examining the intended purpose of each system. Splice is specifically designed for real-time high performance distributed computing. Whereas the Jini technology used by the NID system is designed for ad-hoc networks in a heterogeneous environment.
For a relatively static real-time application with substantial communication and performance requirements, Splice is the best-fit system. This includes our framework application and applications like robot soccer and traffic control. NID is more suitable for applications without real-time constraints where devices connect and disconnect frequently. Home automation, with all sorts of domestic devices that communicate with each other, is such an application.

References

[1] M. Boasson. Architectural Support for Integration in Distributed Reactive Systems. *Proceedings of COMPSAC '01*, IEEE Computer Society, Los Alamitos, 2001.

[2] M. Boasson. Control Systems Software. *IEEE Transactions on Automatic Control*, vol. 38, no. 7, pp. 1094-1107, July 1993.

[3] M. Boasson. Embedded Systems Unsuitable for Object Orientation. *Ada Europe 2002, Lecture Notes in Computer Science*, Springer, to be published.

[4] N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, 1989

[5] B. Clifford Neuman. Scale in Distributed Systems. *Readings in Distributed Computing Systems*, IEEE Computer Society, Los Alamitos, pp. 463-489, 1994.

[6] L. Gasser. MAS Infrastructure Definitions, Needs and Prospects. *Proceedings of the Workshop on Scalable MAS Infrastructure*, Barcelona, June 2000.

[7] Hermann Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, 1997.

[8] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. *Advances in Cryptology – Eurocrypt '90 Proceedings*, New York, Springer-Verlag, pp. 389-404, 1990.

[9] U. Leonhardt, J. Magee. Multi-Sensor Location Tracking. *Mobile Computing and Networking*, pp. 203-214, 1998.

[10] D. López de Ipiña. *TRIP: A Distributed Vision-based Sensor System*. PhD 1st Year Report, Laboratory for Communications Engineering, Cambridge University, 1999. (http://wwwlce.eng.cam.ac.uk/~dl231/trip/docs/report.ps.gz)

[11] K. Mani Chandy, J. Kiniry, A. Rifkin, D. Zimmerman. A Framework for Structured Distributed Object Computing. *Parallel Computing*, vol. 24, no. 12-13, pp. 1901-1922, 1998.

[12] M.G. Maris, J.A.A.J. Janssen, H. Adriani. *The NID Concept*. TNO-FEL internal report, March 2000.

[13] A. Mohindra, U. Ramachandran. *A Comparative Study of Distributed Shared Memory System Design Issues*. Technical Report: GIT-CC-94/35, College of Computing, Georgia Tech, Atlanta, August 1994. (http://citeseer.nj.nec.com/mohindra94comparative.html)

[14] M. Robillard. *Security and Protection: New Trends in Distributed Object Systems.* Literature Survey, Department of Computer Science, University of British Columbia, Vancouver, April 1998.
(http://citeseer.nj.nec.com/robillard98security.html)

[15] Sun Microsystems. *Java Remote Method Invocation Specification*.
(ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf)

[16] Sun Microsystems. *Jini Network Technology*.
(http://www.sun.com/jini)

[17] Andrew S. Tanenbaum. *Computer Networks 3rd edition*. Prentice Hall, New Jersey, 1996.

[18] G. Wells, A. Chalmers, P. Clayton. A Comparison of Linda Implementations in Java. *Communicating Process Architectures 2000, Proceedings of WoTUG-23,* IOS Press, Amsterdam, 2000.