KUNGL
TEKNISKA
HÖGSKOLAN

CID-189 • ISSN 1403-0721 • Department of Numerical Analysis and Computer Science • KTH

A State Sharing Toolkit for Interactive Applications

Mikael Lindkvist

CID, CENTRE FOR USER ORIENTED IT DESIGN

Mikael Lindkvist

A State Sharing Toolkit for Interactive Applications

Report number: CID-189
ISSN number: ISSN 1403 - 0721 (print) 1403 - 073 X (Web/PDF)
Publication date: April 2002

Reports can be ordered from:

CID, Centre for User Oriented IT Design
NADA, Deptartment of Numerical Analysis and Computer Science
KTH (Royal Institute of Technology)
SE- 100 44 Stockhom, Sweden
Telephone: + 46 (0)8 790 91 00
Fax: + 46 (0)8 790 90 99
E-mail: cid@nada.kth.se
URL: http://cid.nada.kth.se

A State Sharing Toolkit for Interactive Applications

Mikael Lindkvist

Today there exist several interesting examples of Networked Virtual Environments, such as collaborative e-learning environments, military simulations and multi player games that have had an outstanding success. Even though these systems serves have been proven useful for their purpose, however, there are not many pieces of the existing applications that are easy to reuse. In this thesis I present a formal model of the data sharing used for interactive applications and compare it to the models used in traditional distributed systems. I present Streep, a state sharing toolkit for interactive applications, which I hope can serve as a building block for the Net-VE developer.

Table of contents

1. Introduction, background and motivation 3
2. Method 8
3. Design issues 9
4. Implementation issues 16
5. Current usage situations 26
6. Future Work...30
Appendix A - Streep programming tutorial...32
Appendix B – Streep reference manual 34
References...73

1. Introduction, background and motivation

Paradigms in distributed computing

I would like to distinguish tree conceptually different ways for computer programs to communicate. For the discussion I use the terms 'computer program' and 'communication' here in a very broad sense. The programs that communicate could be any pieces of code that runs on a computer, and the communication could take place within a single process, between different processes on a single computer or in loosely coupled system on the Internet. However, I call these models *message passing*, *procedure invocation* and *shared storage.* In the simple case where the communication takes place within the same computer, these models can be exemplified in the following way.

A good example of message passing is *pipes* in the UNIIX operating system, where one process can write a stream of bytes in one end of the pipe and another process can read them, in the same order as they were inserted, in the other end. Procedure invocation is when a program makes a call to a library function, such as opening a file or converting a string to an integer. Shared storage is when some pieces of code share a common memory are where someone writes data that someone else reads. An example of this is the framebuffer on graphics hardware where a program writes data representing graphics, and the graphics device displays this graphics on a display. Depending on the underlying environment, it is often easy to implement one of these paradigms for a systems programmer, while the application programmer might prefer a different model. Therefore it is common to implement one model on top of another. A queue data-structure, for example, can be seen as a way to implement message passing on top of shared storage.

Now, the challenge is how to implement these models in an environment where the communication takes place over large physical distances, such as on a LAN or on the Internet. In this environment the message passing paradigm is clearly the most natural to implement, and the TCP/IP protocol suite is an excellent example of such as implementation. There are also many implementations of procedure invocation on top of TCP/IP, such as RPC[1] and CORBA[2]. It is hardly surprising that while the TCP/IP is very general and well suited for most applications, packages with a higher level of abstraction such as those mentioned above usually have different characteristics. Which package to use is often a critical choice for a software-engineer, and depends on the needs of his[*] application. When it comes to implementing shared storage in a distributed environment the design choices are even more diverse. The concept of having shared storage implemented on top of message passing is know as *distributed shared memory* and is an area of research related to distributed systems. Many such systems that have been developed are intended to be used for parallel computations. They are usually designed with robustness as an important goal, and hence they use secure communications and other things

[*] "He" should be read as "he or she" and "his" should be read as "his or her" throughout this thesis.

that impose a large degree of overhead. These characteristics make them less well suited for highly responsive, interactive applications. Streep is an attempt to design a shared memory system that is better suited for such applications.

As a part of the background I will present some algorithms from the distributed systems community that deals with distributed shared memory and discuss some models used for distribution in networked virtual environments. I will also present a case study of a traditional shared memory system and argue that its model is not well suited for interactive applications.

Theories used in distributed systems

Much research has been done on traditional distributed systems. Here I present some of them that are related to my thesis. They are all taken from Tanenbaums book "Distributed operating systems"[3].

Centralized vs. decentralized systems

The distributed systems community often talks about the importance of having *decentralized* systems. The reason for this is to avoid the *single point of failure* problem, i.e. a distributed system should not be designed so that the crash of one participant causes the whole system to stop functioning. The implementation such decentralized systems often impose a large degree of overhead. Also, in reality attempts to implement such systems may sometimes lead to a *multiple point of failure* problem, i.e. the whole system fails as soon as any participant crashes. Therefore, many systems use a central server, or coordinator, to manage the other participants.

Mutual exclusion

Systems involving multiple processes are often most easily programmed using critical sections. When a process wants read or write shared data it first enter a critical region to achieve mutual exclusion and ensure that no other process will access the data at the same time. Tanenbaum presents three methods for achieving mutual exclusion in a distributed system, a centralized algorithm, a decentralized algorithm and an algorithm based on a circulating token.

Figure 2.1 shows a comparison of the three methods when it comes to the number of network messages that need to be transmitted when a participant wants to enter a critical section.

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Potential problems
Centralized	3	2	Coordinator crash
Decentralized	2 (n-1)	2 (n-1)	Crash of any process
Token Ring	1 to inf.	0 to n-1	Lost token, process crash

Fig 2.1: Comparison of models for mutual exclusion

In addition to these numbers, it should also be noted that in the centralized algorithm no messages are needed if the host who wants the lock is the coordinator.

Given these values, it can be argued that for an interactive system where the response time is important the centralized approach is probably the best solution.

Consistency models for shared memory

The concept of a distributed shared memory is an attempt to abstract out the actual communication by making it appear to the applications programmer as if two processes on different machines can access the same memory. Ideally, this shared memory would work exactly in the same way as if the two processes ran on the same machine. However, in reality this is very difficult to achieve which means that the applications programmer need to adapt to certain constraints when it comes to how his program accesses the shared memory, the *consistency model* of the shared memory decides which constraints those are. I will discuss five of the most common consistency models here.

The first one is known as **strict consistency**. Using Tanenbaum's definition, this model can be described as follows:

> *Any read to a memory location x returns the value stored by the most recent write operation to x.*

While this definition seem clear an obvious, it is impossible[4] to achieve in a distributed system because it assumes the existence of an absolute global time. Therefore a slightly weaker model has been formulated, known as **sequential consistency**:

> *The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

This means in practice that while strict consistency stipulates that all participants should see the memory reads and write exactly when they happen, in sequential consistency it is enough if all participants see the reads and writes in the same order. However, if the programmer adheres to reasonable programming practices, this is enough to provide a transparent distributed memory.

While sequential consistency is designed to be transparent to the programmer, one can usually get better performance if one uses a weaker model known as **weak consistency**. This model uses special *synchronization variables*. The role of the synchronization variable is to make sure that all participants have the same contents of their local memories, and as soon as a synchronization variable is accessed the system takes action to ensure this. When a participant wants to access the memory, he first has to lock it by accessing one of these synchronization variables to ensure that he has the most updated version of the memory contents. When he is done reading and writing, he accesses the synchronization variable again to make sure that that all other participants are aware of the updates.

In weak consistency, the memory contents are "brought in" when they are needed by a participant, and "pushed out" when a participant is done with them. One can see that it would be enough to just perform one of these operations if all participants agree on which one. The contents could either be

requested when needed, i.e. on the entry of a critical section, or they could be sent out as soon as they are modified at the end of a critical section. This gives us the two last consistency models, **entry consistency** and **release consistency**. It should be noted that release consistency gives better performance if there are more reads than writes to the memory, while entry consistency has the reverse properties. Since reads are usually more common than writes, release consistency the more commonly used model where performance is critical.

A case study: Linda

The Linda[5] system provides processed on multiple machines with a highly structured distributed shared memory. The concepts of Linda can easily be added to existing programming languages, and several implementations exist for languages such as C and Fortran. The central idea of Linda is a shared abstract *tuple space*. The programmer has a set of primitive operations that can be used to insert, remove and modify the tuples in the tuple space. Each tuple resembles a structure in the C language, in the way that it has a number of fields where each field is of a primitive data type, such as integers, long integers and floating-point numbers. A tuple-field can also hold more complex variables such as arrays and strings. Fig 2.2 shows examples of Linda tuples.

```
("abc", 1, 2)
("counter", 10)
("name-1", "Greg Lim Joon")
```

Fig 2.2: Linda tuples

The operations that can be performed on these tuples are the *out* and *in* operations. The *out* operation is used to put a tuple in the tuple space, as in

```
out("abc", 1, 2)
```

The *in* operation is used to search the tuple space for a tuple matching a given pattern. For example, the operation

```
in("abc", 1, ?i)
```

would search the tuple space for a tuple where first field is a string that has the value “abc” and the second field is an integer with the value 1. The third parameter in this case is not used to do any matching, but instead the contents of the tuple’s third field would be put into the variable *i*. Assuming that an *out* operation like the one above has been performed, *i* would contain the value 2 after the operation. When an *in* operation is performed, and a matching tuple is found, that tuple is removed from the tuple space. If no matching tuple is found, the calling process is suspended until such a tuple is inserted by another process.

The fact that the process blocks this way is central to Linda and is very useful in many cases. It can be used to implement semaphores for critical sections as well as atomic transactions, for example.

The Linda system is built on a neat and simple model and it is easy to see why it is widely used in parallel systems for scientific calculations. However, many concepts that make Linda useful for such systems makes it unattractive for the development of interactive systems. For example, as soon as a

participant wants to know the current value of a tuple, it has to explicitly ask for it, thereby generating a request to the other participants and waiting for a result. There is no method in the Linda model that allows a participant to "push" out a modification to the other participants. The fact that processes are suspended, which is useful for non-interactive systems, is another thing that is unattractive in an interactive system.

Distribution in networked virtual environments

The distribution models used in Networked Virtual Environments are less investigated and formalized than those used in other distributed systems. Michael Zyda points out that while there exist several monolithic Network Virtual Environment systems that have proven to be useful for their purpose, there are not many pieces of those systems that are easy to reuse. Hence,

> *The current focus in the net-VE arena is on the development of toolkits that simplify the development of net-VEs and provide a standard framework for net-VE application development*[6].

Therefore, most Net-VE systems described in literature use a quite low-level approach. The SIMNet[7] environment, developed for the US Department of Defense, is a good example of such a system. The networking architecture in SIMNet is build around highly specialized messages being broadcasted at regular intervals on a network. There are also Net-VE systems that build on traditional ideas from distributed computing, such as the DIVE[8] system, sometimes with low performance and lack of flexibility as a result.

2. Method

Choice of method

As stated earlier, my goal has been to create a useful and intuitive library for dealing with the network related problems related to the development of Net-VEs. I do not want it to depend on external libraries or tools, and I do not want the user to depend on external tools or pre-compilers in order to use the library. I want the library to be accessible from C, with the possibility to extend it to C++ and Java. The underlying principle of the library should be to have a replicated database that can be updated using reliable or unreliable network transport protocols. While these goals and this principle seem clear, there are many theoretical and practical issues that remain. What makes the library intuitive to work with? How do I ensure its usefulness?

I decided to make a list of issues that I needed to solve, and then search for earlier work that I could analyse to find solutions to these issues.

Evaluation

Given the goal of my thesis, the evaluation of the work is best done by looking at its usefulness for an application developer of networked interactive systems. I feel very satisfied to find that today there are several applications that use Streep for their communications in different ways, as described in chapter 6.

3. Design issues

Earlier Work

I found three libraries that in some sense had a similar design to what I would like to create. The first library is the Java Shared Data Toolkit[9] (JSDT) developed by Sun's JavaSoft division. The second one is the CAVERNSoft G2[10] library developed by Jason Leigh et al. at the University of Illinois at Chicago. The third one is the Diverse Toolkit[11] (DTK) developed at Virginia Polytechnic Institute and State University. I compare these libraries with respect to some issues that I feel are important and make them useful and intuitive to work with.

I have also investigated some other libraries, systems and protocols that deal with state sharing and shared memory. Although their goals might not always have been the same as mine, they have many interesting aspects that have inspired me in my work. These include the Virtual Reality Transfer Protocol[12], Repo-3D[13], Shastra[14] and the protocol used in the games Half-Life and Team Fortress[15].

Design issues

Connection and deployment

Since a distributed system by nature runs on several computers, it is often a non-trivial problem in itself how the system should be started. In the case where the system is decentralized, one way to do this is to use a nameserver where all participating applications register as they are started. Hence, each application need to be configured with the address of the nameserver and the nameserver itself needs to be configured. Since this has to be done at the user level, it is often a tedious and error prone task. If the system is centralized this is a bit easier since there is a natural rendezvous point, namely the server, which all applications can contact. This is an argument motivating why my system is centralized, despite the single point of failure problem. I was happy to see that the three other libraries also used a centralized approach.

When I had decided that the system should be a centralized, i.e. have a central server, I had to decide what a "server" actually means in this context. I thought of two possible approaches here. I could either provide functions in the library that let the programmer create a server, or I could provide a generic standalone server program and just provide library functions that let the programmer connect to that program. Both approaches have pros and cons, of course. A standalone server might be easier to get started with for people new to the library, but the other approach offers more flexibility. I found that one of the libraries, DTK, used a standalone server while the other two used the approach to let the user create the server himself. I decided to use a hybrid approach where both provided functions in the library to create a server and also a small program that used those functions to create a standalone server.

Information about participants and security

Streep does not include any mechanism to receive information about the participating hosts. I avoided putting such a mechanism into the library since it

can easily be layered on top of it, for example by having a shared memory block where each participant register themselves by putting their name and ip number.

I also avoided the issue of security. While it would have been easy to implement a simple security scheme using password identification and per-block access control lists, for example, such a scheme would not have been very secure unless it would have been used together with encryption, which I feel is beyond the scope of my thesis.

Event processing

As all network programming is somewhat asynchronous in nature, the issue of how to handle the event processing in an intuitive way is very important. I saw two different main directions of approaching this problem. The first approach was to make the library multi threaded internally, and to let a separate thread handle all the incoming network traffic. The second approach was to include an event processing function that the programmer has to call at regular intervals to process the incoming messages. The libraries that I analyzed used both approaches; JSDT and DTK are internally multi threaded while CAVERNSoft uses explicit event processing methods. I decided to use the latter of the methods since I did not want to make my library dependent on an external threading library. Having made this design decision a mechanism to deal with diversity in bandwidth and computer power was needed. How can one deal with the situation that one participant has a fast processor that sends a lot of data while another participant has a processor that is not fast enough to deal with all the data? Using the approach with an explicit event processing function, this could give rise to two unfortunate scenarios. Firstly there is a possibility that the processing function is not called often enough by the receiving host, leading to a growing queue of unprocessed network messages. In an attempt to avoid this possibility one might be tempted to always process all the queued network messages. This is not possible, however, since the queue might grow as a message is processed, leading to the situation where the application is stuck doing nothing else than processing network packets. To aid the programmer to avoid these situations, I let the event processing function take two parameters, one *wait timeout* parameter and one *read timeout* parameter. The wait timeout parameter specifies the maximum time to spend waiting for data to arrive. If no data are received within that interval, the function will return without doing anything. If data are received, the function tries to receive all that data, but it does not spend more time than specified by the read timeout parameter doing so.

Creating and disposing memory

The straight forward method for creation and deletion of memory blocks is to have one function, *malloc*, that allocates a memory block on the heap, and another one, *free*, that is used to free memory on the heap. This works well on a system that is supposed to run in a single thread on a single computer. In a distributed system where several participants can allocate and free memory, however, things are more complicated. Consider, for example, the scenario where several participants have requested references to the same shared memory block, and one participant decides to free that memory. How should this event appear to the other participants? One option is to have the memory

block removed from all participants' memories as soon as one participant frees it. Using this method there might be a possibility that a participant is not aware of the fact that another participant has deleted the memory block and therefore continues to access the memory, most likely leading to a page fault. Another option is to associate an internal counter with each memory block. Each time a participant request a reference to the block, this counter is incremented by one and each time a participant frees the block the counter is decremented. While this scheme is more robust, it still has problems since each memory block has a name associated with it. If a participant frees a memory block with a given name, he would expect it to be possible to allocate another block with the same name. In the scheme where counters are used, this would not be the case. To solve this issue I introduced a concept of valid and invalid blocks. When a participant frees a block it is invalidated, but not entirely removed from memory. When a block is invalidated, it is not possible to get new references to the block, but participants that already have references can still continue to use the block. All participants will also get an event message so that it is possible to see if another participant has invalidated a memory block.

Getting reference to existing shared memory

An issue related to the creation and disposal of shared memory blocks is the method used to get a reference to an existing block. I saw one potential problem here, namely that when a participant requests such a reference he would expect the memory block to have valid data content. This may not be the case if the block is newly allocated and the participant that created it has not yet written any data there. In this case the contents of the memory would be garbage, something that might be difficult to detect by the requesting participant since there are no general way to distinguish valid data from garbage data. I therefore added the possibility to lock the memory as it is allocated, and also functionality so that the requesting participant can not get a reference to the block until it is unlocked.

Local reference

There are several possible options when it comes to how a local reference should look to the programmer. It could be an opaque object, an integer, or a text string. All of these require explicit read and write operations to access the data. It could also be a simple memory pointer, enabling the programmer to access the data in the same way as local memory. All methods have different pros and cons. To access the data as local memory is convenient, but it makes references to other blocks difficult. A memory pointer that points to an address in the memory of one participating host is not very useful to another host. To access the data using integers might lead to name-space problems. If one host creates a block and gives it the key 1234, for example, another host might want to create a block with the same key and be confused when it is not possible. Text strings are better since they automatically have some meaning coded in to them. If a host creates a block called "avatar_positions" when it deals with the creation of avatars for a shared virtual reality system, it is less likely to conflict with blocks that are created for other purposes. However, text strings take more space than integers and are variable in size, which make them more difficult to deal with efficiently.

Given the different characteristics of the different method it is probably a good idea to use combinations to get good performance and at the same time a system that is intuitive to work with. This is the case in the in the libraries that I have analysed. JSDT uses a combination of opaque objects and text strings. When an object is created it is given a textual name that serves as an identifier which is used to get references to the object from other hosts. The access to the data is made using the object ByteArray. The functions that are used to create and get references to shared objects both return an instance of such an object. A similar method is used by DTK. When memory is allocated it is given a name that is used as an identifier to get references to the memory. The access to the memory is made using a dtkSharedMem object. CAVERNSoft uses a slightly different approach. Here the programmer has to explicitly supply a name each time he wants to access an object.

I decided to use a combination of text strings, integers and memory pointers to refer to the objects. When objects are created the programmer supplies a text string that serves as an identifier for the memory block. This identifier can be used to get a reference to the block from another host. The function that creates a block and the function used to get a reference to an existing block both return a memory pointer that can be used to access the block's data. When a block is created the system assigns a 32-bit integer that is not visible to the programmer. This integer is used internally to identify the block when it is sent in network packets.

Sending and receiving updates

The issue of how updates should be sent and received is an important transparency-efficiency trade off that very much affects the usefulness of a shared memory system. In traditional systems transparency has been very important, but in systems where the shared memory is updated at interactive rates a programmer would probably favor efficiency and control over transparency. Analyzing the aforementioned libraries I found that they used a similar philosophy, which lead to the decision of having an explicit notification function that the programmer calls each time a memory block is changed and its contents should be sent to the other participants.

Protocol selection and subscription

Two related issues are the methods used to select what underlying protocol that should be used to communicate memory block updates, i.e. reliable of unreliable protocol, and how a participant should choose what memory blocks it is interested to receive updates for. I decided to solve both these issues by a concept called channels, inspired by concepts found in the JSDT. When the server is created it is passed information about what channels that should be used. The current implementation includes a reliable channel implemented on top of TCP, and two unreliable channels implemented on top of UDP, one using point-to-point communication and one using multicast. The programmer can choose any number of channels to be set up, and each participant can choose what channels it is interested in. These channels can be used both for protocol selection and subscription in an application specific manner. In a shared virtual environment, for example, separate channels could be used for each room in the environment. As a user enters a room, his client would subscribe to the channel for that room. The channels can also be used to deal

with variations in bandwidth. An application could open two channels to be used for the same data, for example to deal with avatar position updates. One of the channels could be used to send updates 2-5 times a second while the other one could be used to send updates at a higher frequency, maybe 50 times a second. The application could then be configured to listen to different channels depending on if the user has a modem or a local area network connection.

Consistency control

While consistency control is often considered to be the most important issue in many distributed shared memory systems, it can be argued that it is not quite as important for a library that has the same intended application areas as Streep has. However, there are possible situations where the programmer would like consistency to be ensured. Such a situation could be the maintenance of a list of participants for a shared application. For those situations I included mutual exclusion locking functionality, with the possibility to explicitly lock and unlock shared memory blocks, thereby effectively implementing release consistency.

Serialization

If two participants run on machines with different byte order, such as a Macintosh and a PC, care has to be taken that data structures are correctly serialized. In C and C++ there is also another potential problem since structures are aligned for better performance. The structure in figure 3.1, for example, contains 5 bytes of data but the size in memory is usually 8 bytes since the integer is aligned to a four byte boundary. How structure fields are aligned might also differ depending on the compiler. I would like to have a mechanism for the programmer to describe the data-structures used so that the system can do this serialization as transparently as possible.

```
struct s
{
   char c;
   int i;
};
```

Fig 3.1: A structure in the C language

Java has ways to obtain information about what fields an object contains, using the `.getClass()` method and so on, which makes this serialization really neat and transparent. Hence, JSDT does not have this problem. Unfortunately things are not that simple in C and C++. Therefore the CavernSoft library has a special datapack class, that lets the programmer create an object and then pack each field of a structure into this object. When all fields are packed, the programmer can call a method and obtain a pointer to the serialized data. The DTK leaves the structure serialization entirely to the programmer. Neither of these schemes appealed to me due to their lack of transparency, so I searched for other methods to do this.

The Datareel[16] system, developed by glNET software, uses a concept called Interoperable Data Types. The concept is based on C++ and uses special classes that resembles the ordinary built-in datatypes, like vbINT32 instead of int. The vbINT32 datatype is a class that contain an array of four characters and have all arithmetic and assignment operators overloaded to pack a value into the character array. This way it is possible to use a vbINT32 variable in the same way as an int variable, but the byte ordering will always be the same regardless of the underlying architecture. This method also overcomes the structure alignment problem since characters are just one byte long. While the

concept is neat and transparent, I rejected it mainly because of two reasons. Firstly it depends on C++. Secondly, which is more important, is that even though it is possible to use a vbINT32 variable in the same way as an int variable, it is not possible to use a vbINT32 array in the same way as an int array. It would not be possible, for instance, to pass a pointer to vbFLOAT array to the OpenGL function `glVertex3f()` which expects a pointer to a float array.

```
c - char (8 bit)
s - short (16 bit)
l - long (32 bit)
f – float
d - double
```

Fig 3.2: Structure serialization characters

There also exist more standardized methods to do structure serialization. One such method is the Sun XDR[17]. In this case the programmer writes an external representation of his structures in a special language that resembles C. Then he runs this file through a XDR compiler that generates platform-specific C code to serialize and unserialize the structures. I rejected this method because I did not want my system to rely on external tools or precompilers because these are troublesome to integrate with the integrated development environments commonly being used today.

The method I finally settled with was to let the user specify the data-format by supplying a string at the creation of each block. The string contains a sequence of characters like in figure 3.2. For more information about the internal workings of this method and how I solved the byte-ordering and alignment issues, see chapter 5.

I later found that this method is similar to a method used by the Transarc De-Light[18] library by IBM.

Late arrival

When a participant joins a session, there might be large amounts of data residing in the database on the server. In order to avoid long initial waiting times as a client connects, the client has to explicitly request references to all data blocks that it is interested in. Similar approaches are used in the other investigated libraries.

References and hierarchies

Something that one often wants to do is to have one block of memory referring to another block, in the same way as one often has a pointer in one structure that points to another structure. This would be useful, for example, in a shared VR system where each participant is represented by an avatar. In this case each participant needs to know the positions of the other participants, in order to render the participants graphical representations. This could be implemented by having a shared memory area that contains an array of references to other areas, each representing the data related to one avatar. As a participant joins the system, his client program inserts a reference into the list. When he leaves, the program removes that reference.

One elegant and transparent scheme for dealing with such references is *pointer swizzling*[19]. I first read about this technique in the context of the InterAct[20] system, developed at the University of Rochester. The technique works by converting object pointers to an external representation, for example

32 bit integers, before an object is sent over the network. When the object is received this external representation is converted back into an ordinary object pointer. This way all references will look like ordinary pointers to the programmer, which will be both efficient and convenient. The system needs mechanisms, such as hash tables, to convert pointers to external object representations and back again.

Since the local reference of an object in my system is a pointer, this scheme would fit pretty well into the model. At one point in time I also had a mechanism for pointer swizzling implemented. However, I later decided to remove it due to its error prone attributes. Since objects can be subscribed and unsubscribed to dynamically there might be a possibility that a client might receive a reference to an object without having the object, which would lead to confusion. Instead references from one object to another have to be more explicit, for example by including the object's name.

4. Implementation issues

Given the concepts in the previous chapter many practical implementation issues remain. In this chapter I discuss the different decisions I made regarding this implementation. My implementation exposes a C API to the programmer containing some 30 functions. Even though the API is in C, the actual implementation is made in C++, mainly to get access to the standard template library[21] (STL) and its efficient dynamic array and mapping classes. Before I will describe my implementation class by class, I would like to describe some concepts that are central to many of the classes.

Concepts

Local reference as pointers

As mentioned in the previous chapter, a local reference to a shared memory block appears to the user as an ordinary memory pointer in the same way as a pointer returned by the standard `malloc` function. However, even though the user sees and accesses this memory in the same way as memory allocated by standard C routines, the system needs to know more about this memory. For example, if the programmer issues a `srNotify(p)` call, where `p` is a reference to a shared memory block, the system needs to know the size and the internal key for that memory block. For this purpose the block allocated internally is actually a bit larger than what the user requests, and this extra information is stored *behind* the pointer returned to the user. An allocation request for a block of say 20 bytes would result in a scenario like that in fig 4.1.

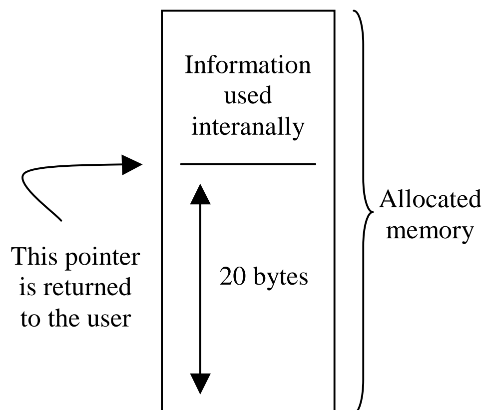

Fig 4.1: Local reference to shared data

Asynchronous I/O

In low-level network programming using TCP/IP, one has the option of using either *blocking* or *non-blocking* sockets. The difference is that when reading from a blocking socket, using the `recv` socket API call, the calling process will be suspended until data arrives as opposed to the non-blocking mode of operation where a `recv` call will have the effect of polling the socket for data. In traditional, command line UNIX programs, such as *ftp*, *telnet* and *rcp*, the best option is probably to use blocking sockets since the behavior closely resembles standard UNIX file I/O. In interactive, graphical applications, however, the behavior of having the process suspended is not very attractive and therefore it is possible to use non-blocking sockets and poll for incoming data at regular intervals. The problem with this approach is that it is not very operating system friendly, especially if we have many sockets where we expect

incoming traffic. A friendlier alternative is to use blocking sockets and the `select` API function, which has the effect of waiting for traffic from several sockets at once. For more information on socket programming, see [22].

All classes that deal with networking implements the two methods

- `GetRelatedSockets(fd_set *set)`, and
- `Process(fd_set *set)`

where the `fd_set` data type is the same as is accepted by the `select` socket API function call. This was it is possible to use blocking sockets, and just make one `select` call in the `srProcess` function that can wait for incoming network traffic in an operating system friendly way, even though there are many classes that are interested in listening for network traffic.

Blocking and non-blocking operations

The `srGetMem` and `srLock` functions, resulting in `getrequest` and `lockrequest` messages, accept the optional flags `NOBLOCK` and `NOQUEUE`. These flags control the behavior to be taken when a client requests a reference or lock for a memory area that is locked by another client.

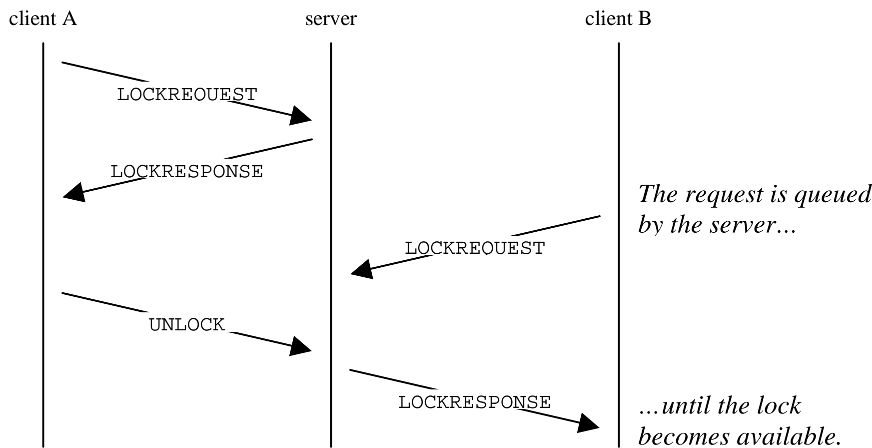

Fig 4.2: Default behavior if two clients request a lock for the same block

The default behavior is for the requesting client to block until the client that holds the lock has released it. the messages that would be passed between the server and the client in this scenario can be seen in figure 4.2.

If the `NOBLOCK` flag is used, however, the client will not block but will receive an error indication from `srGetMem` or `srLock` function and the request will be queued in the server. When the lock later becomes available, the requesting client will receive an event about this, like in figure 4.3.

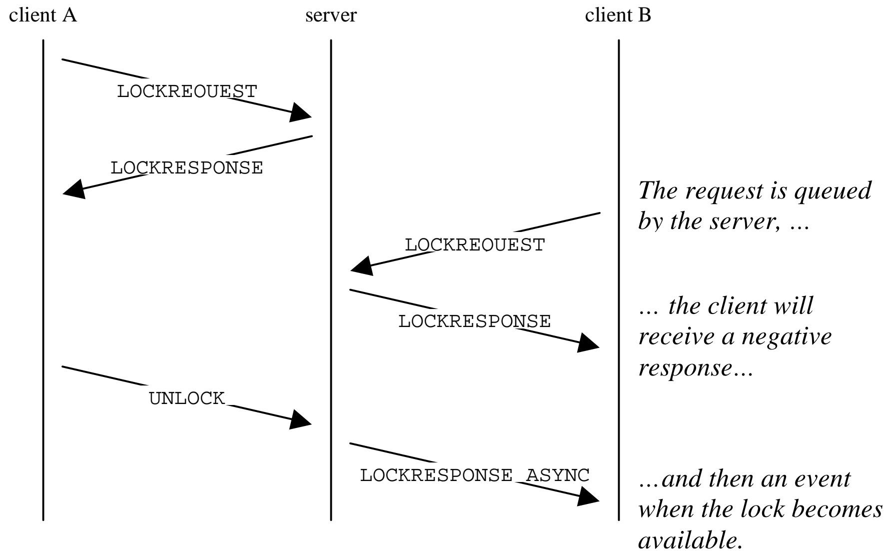

Fig 4.3: NOBLOCK behavior if two clients try to lock for the block

The `NOQUEUE` flag works in the same way as the `NOBLOCK`, with the distinction that the request will not be queued; if the lock can not be obtained immediately it will not be obtained at all. The messages that are passed in this scenario are shown in figure 4.4.

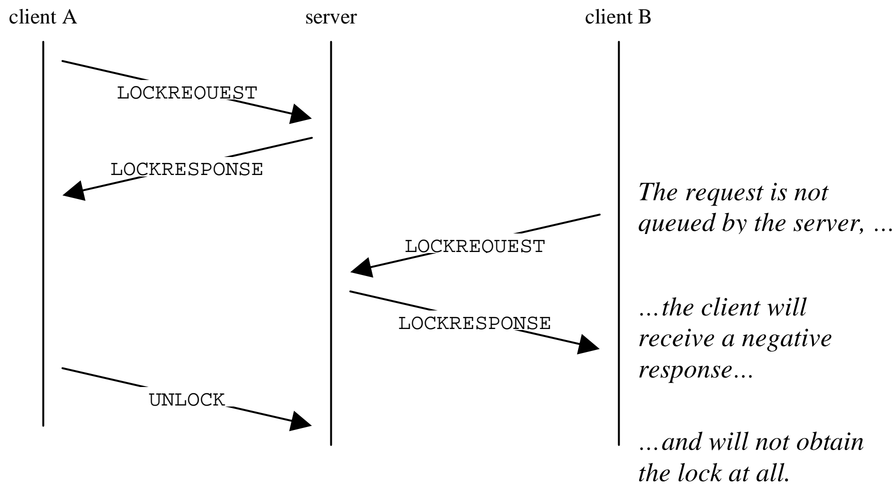

Fig 4.4: NOQUEUE behavior if two clients try to lock the same block

Control protocol

The control protocol used through the `SRConChannel` class consists of a set of network messages. Each message is built up using a number of fields where each field is one of the possible types 32-bit integers, null terminated strings or raw data. The messages can roughly be divided into two groups; messages sent from a client to the server and messages sent from the server to a client. The possible messages are described below.

CHANNELINFOREQUEST

The `channelinforequest` message is sent from the client to the server to request information about the channels beeing used.

CHANNELINFORESPONSE
int *numchannels*
string *channeltype 0*
string *channelid 0*
string *channeltype 1*
string *channelid 1*
. . .

The `channelinforesponse` message is sent from the server to the client to inform it about what channels are used. The message is sent as a response to the channelinforequest message.

LOCKREQUEST
int *key*
int *flags*

The `lockrequest` message is sent from the client to the server to request a mutual exclusion lock for a shared memory block.

LOCKRESPONSE
int *key*
int *blockdatasize*
raw *blockcontents*

The `lockresponse` message is sent from the server to the client as a response to a `lockrequest` message. The *key* field is zero if the lock could not be obtained.

LOCKRESPONSE_ASYNC
int *key*
int *blockdatasize*
raw *blockcontents*

The `lockresponse_async` message is sent from the server to the client as a response to a `lockrequest` message if lock request was queued.

ALLOCREQUEST
string *name*
int *size*
int *flags*
string *format*

The `allocrequest` message is sent from the client to the server to request the allocation of a new block. The *format* field contains a data description string used in the `SRSerializer` class.

ALLOCRESPONSE
int *key*

The `allocresponse` message is sent from the server to the client as a response to the `allocrequest` message. If a new block could not be allocated, i.e. if a block with the requested name already exists in the repository, the *key* field is zero.

GETREQUEST
string *name*
int *flags*

The `getrequest` message is sent from the client to the server to request a reference to a block in the repository.

GETRESPONSE
int *key*
int *size*
string *format*
raw *contents*

The `getresponse` message is sent from the server to the client to as a response to the `getrequest` message. If the request failed, i.e. no block with the given name exist in the repository, the *key* parameter in zero and the rest of the fields undefined.

GETRESPONSE_ASYNC
int *key*
int *size*
raw *contents*

The `getresponse_async` message is sent from the server to the client as a response to the getrequest message if the getrequest was queued.

JOINCHANNEL
int *channelid*
string *localpeer*

The joinchannel message is sent from the client to the server when a client wants to listen to a communication channel. There is no response from the server related to this message.

PARTCHANNEL
int *channelid*

The partchannel message is sent from the client to the server when a client does not want to listen to a communication channel any more. There is no response from the server related to this message.

RELEASE
int *key*

The release message is sent from the client to the server when a client releases a reference to a shared memory block, as a result from the seReleaseMem API call. There is no response from the server related to this message.

FREE
int *key*

The free message is sent from the client to the server when a client deletes a shared memory block from the repository as a result from the srFreeMem API call. There is no response from the server related to this message, but the message will lead to an invalidation of the memory block, resulting in an invalidate message being sent to all clients that currently holds a reference to the block.

UNLOCK
int *key*
int *size*
raw *contents*

The unlock message is sent from the client to the server when a client releases a mutual exclusion lock held on a shared memory block. There is no response from the server related to this message, but the message may result in

a `lockresponse` or `lockresponse_async` message being sent to another client if there are queued lock requests for the block.

INVALIDATE
int *key*

The `invalidate` message is sent from the server to the client as a result of an invalidation process, i.e. if a client has deleted the block from the repository.

Classes and modules

The implementation is made up by the classes in figure 4.5. I will here describe the implementation class by class.

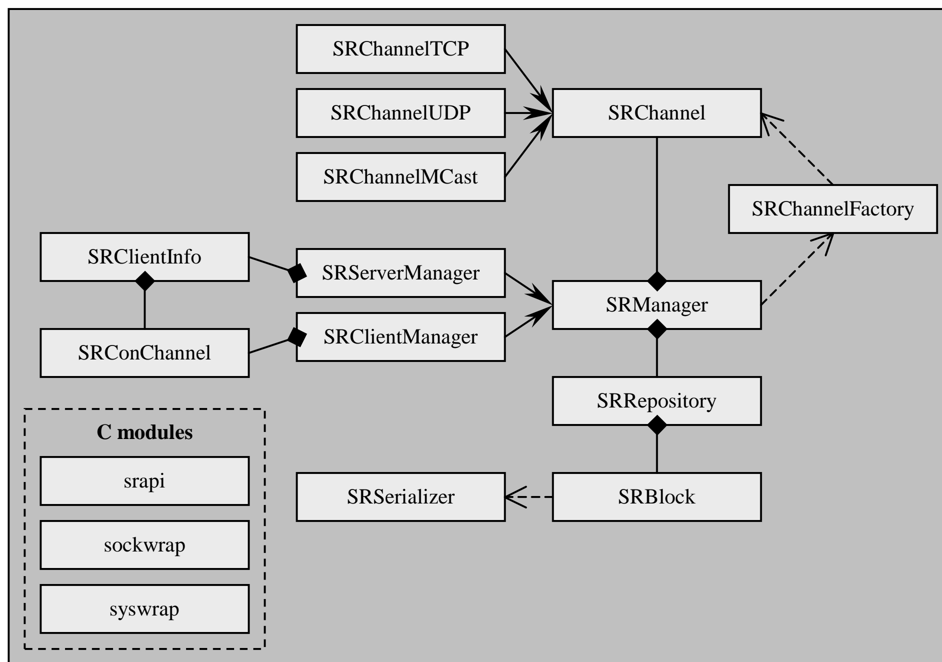

Fig 4.5: Implementation class diagram

srapi

The srapi module contains the exposed user functions. This module contains one data type, `SR`, which is used as an opaque object identifying repository connections. Internally this object is actually an instance of the `SRManager` class. Most API functions does little else than cast a `SR` pointer to a `SRManager` pointer and then call a class method. An exception is the `srProcess` API function, which operates according to the scheme previously described under Asynchronous I/O.

syswrap

The syswrap module contains a number of C functions used to ensure platform independence of the rest of the code. I have tried to keep the rest of the source code UNIX conformant, so this module basically contains wrappers to Windows functions where Windows and UNIX differ, like the `gettimeofday` function.

sockwrap

The sockwrap module contains functions to keep all socket operations patform independent, as well as a couple of utility function to keep the rest of the code neat and tidy since raw socket programming usually looks quite messy.

SRManager

The `srManager` class serves as a base class for the functionality shared by both a server and a client. Most functions, such as allocation and freeing of shared memory, can be done on both the server side and the client side. Even though the effect appears to be the same to the user, the implementation is quite different in the different situations. When a client wants to allocate memory it has to send a request to the server while the server can immediately go ahead and allocate the memory, for example. Therefore this class contains virtual functions for most functionality provided by Streep, while the actual implementation resides in the `SRServerManager` and `SRClientManager` classes.

SRServerManager

The `SRServerManager` class contains server versions of the methods prototyped in the `srManager` class. This class also contains a vector of `SRClientInfo` instances to keep track of the clients that are connected to the server.

SRClientManager

The `SRClientManager` class contains server versions of the methods prototyped in the `srManager` class. This class also contains an instance of the `SRConChannel` class for communicating with the server.

SRClientInfo

The `SRClientInfo` class contains information about a client that is connected to the server, such as the client's ip-number, and an instance of the `SRConChannel` class to communicate with the client.

SRConChannel

The `SRConClient` class contains an implementation for a control protocol used for client-server communication. The class has an inbound and an outbound queue for messages and methods for packing data into messages and flushing the queues. Each message consists of a 32-bit integer identifying the message type, followed by the size of the message and the message's fields. The possible types of the field are 32-bit integers, null terminated character

strings and raw data. When raw data is packed into a message, there is no size information packed together with that data, so this information is assumed to be known by the receiver, possibly contained in a field earlier in the message. To illustrate the workings of this class, I included a simplified version of the code that packs and unpacks a GETRESPONSE message if fig 5.5. Note that the PackRaw, UnPackRaw and UnPackString methods does not actually do any memory copying but rather they increment the current message size and return a pointer where the data is to be inserted into the message buffer. The memory copying is left to the programmer for efficiency reasons.

```
SRConChannel *con;
SRBlock *block;
---------------------------------------------
con->BeginOutMessage(SRCON_GETRESPONSE);
con->PackInt(block->key);
con->PackInt(block->size);
con->PackString(block->format);
memcpy(con->PackRaw(block->size),block->size);
con->EndOutMessage();
---------------------------------------------
con->GeginInMessage(SRCON_GETRESPONSE);
block->key=con->UnpackInt();
block->size=con->UnpackInt();
strcpy(block->format,con->UnpackString());
memcpy(block->data,con->UnpackRaw(size),size);
con->EndInMessage();
```

Fig 5.5: Packing and unpacking network messages

SRRepository

The SRRepository class contains the local repository replication and is responsible for the creation and deletion of SRBlock instances for each shared memory block in the repository. This class has two STL map structures to get references to the blocks, the first one is a <int, SRBlock *> map to look up blocks by key and the second is a <string, SRBlock *> map to look up blocks by their names.

SRBlock

The SRBlock class contains information about each block in the repository. The class has a vector of SRClientInfo pointers used by the server to keep track of what clients that have obtained references to the block, and a deque of SRLockInfo instances used for queuing lock requests. These fields in the SRBlock are only used by the server.

SRSerializer

The SRSerializer class deals with serialization and deserialization of block contents to be sent over the network. The constructor of the class accepts a format specification string as described in the documentation of the srAllocSerializedMem function in appendix B, which is parsed to an internal byte code representation. The Serialize and UnSerialize can then be used to convert data to and from a platform independent format.

SRChannel

The SRChannel class serves as a base class for the different types of channels used for block updates. The class contains code for packing data in and out of blocks, queues for incoming and outgoing network messages and methods for creating such messages. The actual code to send those packets is

implemented in the various channel implementations that inherit from this class.

SRChannelFactory

A factory to create `SRChannel` instances of different types.

SRChannelTCP

A channel implementation using reliable communication on top of TCP.

SRChannelUDP

A channel implementation using unreliable communication on top of UDP.

SRChannelMCast

A channel implementation using unreliable communication on top of UDP and multicast.

5. Current usage situations

Since the goal of my thesis has been to create a useful toolkit for the development of networked interactive applications, and since a large amount of work has focused on designing a useful and intuitive API, the evaluation is best done by looking at the applications that has been developed using Streep. I feel very satisfied to find that today there are several such applications and I will describe some of them here as well as describe in what way they have used Streep.

Shared nodes in the VRT toolkit

The Virtual Reality Toolkit (VRT) is a scene-graph library developed by Stefan Seipel at Uppsala University. An extension to this library was added using Streep that allowed a user to create shared scene-graph nodes. The standard way to create nodes in the VRT is to use the function

```
VRT_Node *VRT_NodeNew(VRT_Node *parent, char *name)
```

To add networking functionality the following function was added to the library:

```
VRT_Node *VRT_SharedNodeNew(VRT_Node *parent, char *name)
```

Once such a shared node has been created, it will share its local transformation matrix with the other nodes on the network that has the same name, through the use of a Streep shared memory block. The traditional VRT functions for manipulating the matrix, such as `VRT_NodeSetTranslation` and `VRT_NodeSetRotation`, updates the shared memory and schedules it for delivery to the other participants in the network.

The concept of shared nodes provides a simple, general and powerful mechanism to rapidly prototype networked VR applications. It is arguably also an efficient mechanism, since the user has explicit control over which nodes that are to be shared. It has been used for several small demonstration applications at Uppsala University, as well as in the course Interactive Graphical Systems, where students used it in a one week project create several interesting applications, of which two are shown in fig 5.1. The left picture shows a networked battle tank simulator, courtesy of Theresa Carlstedt-Duke, Christoffer Eriksson and Patrik Lakhsasi, and the right picture

Fig 5.1: A networked battle tank simulator and a networked soccer game

shows a networked soccer game, courtesy of Jonas Eklund and Tony Gunnarsson.

CVEL

The Collaborative Virtual Environments for Learning[23] (CVEL) is a subproject within the research framework of the Wallenberg Global Learning Network[24]. The project has focused on developing and testing an integrated virtual environment that serves as a meeting point for students and teachers to meet for collaborative experiments. Streep was used in this project to for data sharing among the participants in the virtual environment. As a part of the project, an experiment was conducted where 16 students participated in a virtual lecture being held in the environment. The results from this experiment showed that that this virtual lecture environment functions as a useable alternative to a traditional lecture theatre. The experiment also showed, which is more important in this context, that Streep can be used to develop applications for a real world scenario with several users logged in at the same time.

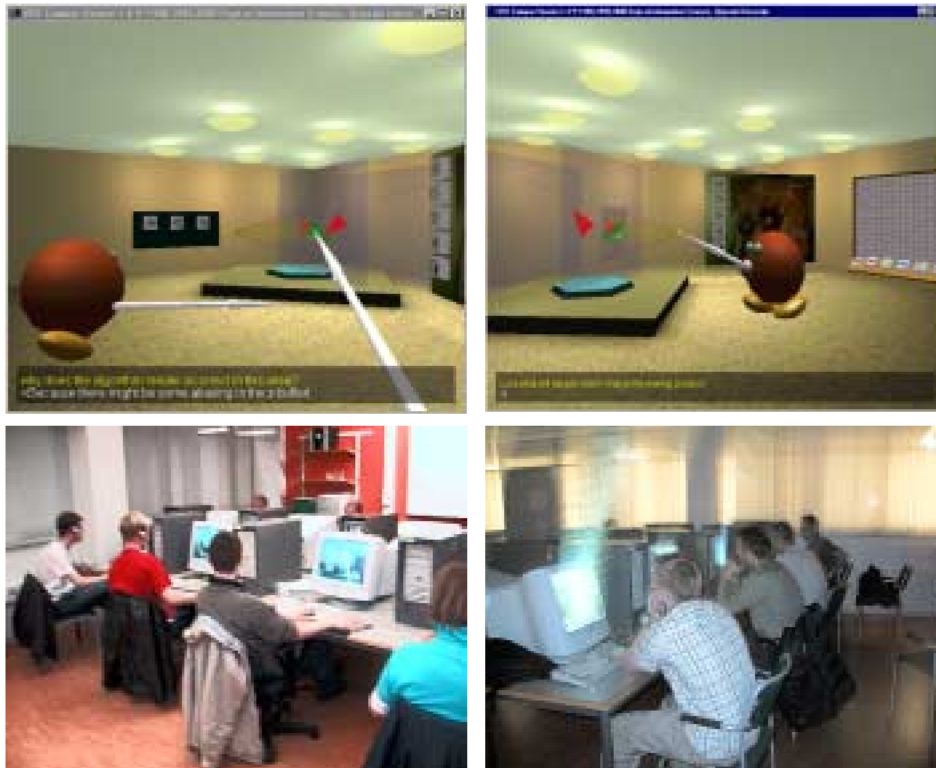

Fig 5.2: The CVEL experiment being used during a virtual lecture

The Aquarium

The Aquarium[25] project, conducted by the Swedish National Defence College, focuses on developing and testing new concepts in the next generation military command and control systems. Streep was used for the communication between various VR-input devices and a distributed rendering system for

Fig 5.3: Experimental set-up for the Aquarium project

displaying and interacting with strategical data. For this purpose a layer called CONVIDES[26] was put on top of Streep for managing virtual device pool, where applications could connect to read device data. Fig 5.2 shows the Aquarium experimental set-up as well as a demonstration application used there. The environment consists of 17 PCs connected together on a local area network. Two PCs control the "visionarium", the large horizontal screen seen on the left image, four PCs serve as "visioscopes", the four large back-projected screens of which one is visible behind the visionarium, while the other PCs serves as workstations for the participants in the session. The purpose of the experiment was to test networking, rendering and interaction approaches that enabled an application to be run in this distributed environment.

High speed craft navigation

The Human Computer Interaction department of Uppsala University has recently conducted a project focusing on human-machine interaction and safety aspects when controlling high speed crafts[27]. One of the experiments in this

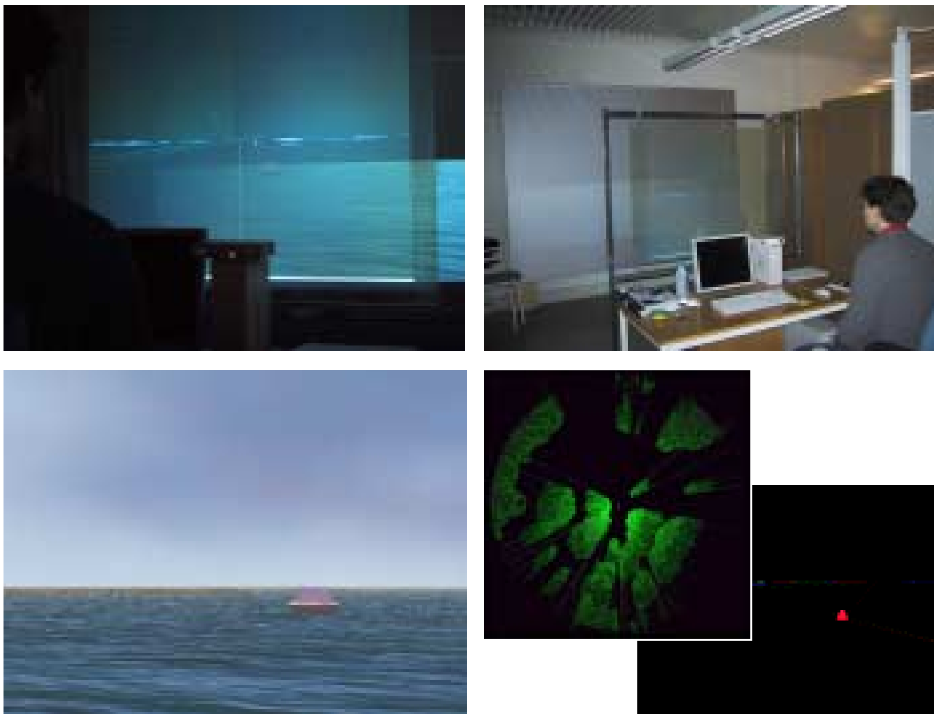

Fig 5.4: High speed craft navigation experimental set-up

project used an augmented reality display to show safety-critical navigation information in the navigator's field of vision.

Streep was used in this project for the communication between the components in the experimental set-up, including a projected simulation environment, the augmented reality display and a simulated radar image. Fig 5.4 shows the experimental set-up (top), screen shots from the simulated environment (bottom left) and the augmented information and simulated radar image (bottom right).

6. Future Work

Even though the current implementation of Streep has many possible areas of usage, there are some improvements that I have thought about but not yet have had time to implement. In this chapter I describe those improvements.

Network time protocol

To save bandwidth or to increase the display update frequency in networked virtual environments it is common to interpolate the positions of shared objects that move. A common method for doing this is to use the dead reckoning[28] algorithm. To use this algorithm, as well as other algorithms used for interpolation and prediction of object positions, one needs to know the update time of the shared data that is to be interpolated. Streep does not currently have any support for this, mainly because sending a timestamp with each network message would not make any sense, since there exist no common time shared by all hosts. One possibility to add such functionality would be to use the Network Time Protocol[29] (NTP). In this scenario the server could act as a NTP server and the other participants as NTP clients, in which case it would be possible to have a common concept of time shared by all participants, with a drift of approximately 10 ms on a local area network according to independent sources on the `comp.protocols.time.ntp` newsgroup. This drift is small enough to make the shared concept of time useful in interactive applications. To expose the functionality to the programmer, the following functions could be added to the Streep API:

- `int srGetTime(SR *sr)` to get the current time in milliseconds, and
- `int srGetUpdateTime(SR *sr, void *p)` to get the last known update time for a shared memory block.

Thread safety

Streep is currently not safe for multi threaded programming, due to global states kept for managing network message queues. Since VR application are often event driven, for example if they deal with external input devices, it would be useful to rethink the message queuing strategy so that it could be made thread safe.

Security

To make Streep secure many things would have to be done to both the network protocol and the current implementation. I am not a security expert, but the things that would be needed includes at least the following:

- User and password authentication.
- Per-block acces control lists, i.e. who is allowed read and modify the contents of the shared memory blocks, and who is allowed to create and delete blocks.
- If one really wants to prevent the data from being read by potential intruders, the block updates would have to be encrypted, for example by using the Secure Sockets Layer[30] (SSL).

- The control protocol that Streep uses today would have to be rethought, and the code would have to be checked for security flaws, such as broken boundary checks.

Better standard compliance

The current design and implementation of Streep is built from scratch, only using the TCP / IP suite protocols for its communication. A more attractive situation would be to have Streep based on some sort of open standard, a standard specifically designed for data sharing in interactive applications. The reason why this was not done is that no such widely adopted standard currently exists. There are, however, efforts with the goal of designing such a standard currently going on. The International Telecommunication Union has created a recommendation for a Multipoint Communication Service, known as the ITU-T.122 recommendation[31]. An interesting topic for further research is to investigate how this recommendation could be implemented and used in the field of networked virtual environments.

Higher abstraction layer

Even though Streep is a useful tool for the Net-VE developer, its approach is still on a quite low level. A higher level abstraction level built on top of Streep has been mentioned previously in the context of shared nodes in the VRT scene graph library. However, this approach has its limitations, and therefore the issue of how such a layer should ideally be created provides an interesting area for further research.

Appendix A - Streep programming tutorial

The code on the following page is a simple program that shows how to use the Streep API. The program can act as either a server or a client. To start it in server-mode, use for instance:

```
example 1200
```

And to start it in client-mode use:

```
example localhost 1200
```

If the program is started as a server, it creates a shared memory area and puts some data there. If it is started as a client, the program will continuously write data into the shared memory block.

```
#include "streep.h"
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  SR *sr;
  int *mem;

  /**** Check arguments. ****/
  if (argc!=3 && argc!=2)
  {
    fprintf(stderr,"Usage: %s <port> for server-mode, "
                   "or %s <host> <port> for client-mode.\n",
                   argv[0],argv[0]);
    exit(1);
  }

  /**** If server-mode, create a server and some memory. ****/
  if (argc==2)
  {
    /**** Use one udp-channel. ****/
    sr=srCreateServer(atoi(argv[1]),"udp");

    if (!sr)
    {
      fprintf(stderr,"Unable to create server.\n");
      exit(1);
    }

    /**** Create a block of memory. ****/
    mem=srAllocMem(sr,"hello",sizeof(int),SR_LOCK);

    /**** Initialize and unlock. ****/
    (*mem)=0;
    srUnLock(sr,mem);
  }
```

```
  /**** If client-mode, create a client and get a reference to the
        block. ****/
  else
  {
    sr=srCreateClient(argv[1],atoi(argv[2]));

    if (!sr)
    {
      fprintf(stderr,"Unable to create client.\n");
      exit(1);
    }

    /**** Get a reference to the block. ****/
    mem=srGetMem(sr,"hello",SR_NONE);
  }

  /** The following code is the same for the server and the client. **/
  srChannelSubscribe(sr,0);

  while (1)
  {
    /**** Change the contents of the memory. ****/
    (*mem)++;
    printf("value: %d\n",*mem);

    /**** Send it onto channel 0. ****/
    srNotify(sr,mem,0);

    /**** This is where the actual network traffic takes place. ****/
    srProcess(sr,SR_NO_WAIT,100);

    sleep(1);

    /**** Have we lost our connection? ****/
    if (srGetError()==SR_NO_CONNECTION)
    {
      fprintf(stderr,"Network error.\n");
      exit(1);
    }
  }
}
```

Appendix B – Streep reference manual

srCreateServer **35**
srCreateClient **37**
srClose **38**
srAllocMem **39**
srAllocSerializedMem **41**
srFreeMem **43**
srGetMem **44**
srIsServer **46**
srReleaseMem **47**
srChannelSubscribe **48**
srChannelUnSubscribe **49**
srChannelGetInfo **50**
srProcess **51**
srLock **53**
srPUnLock **55**
srUnLock **56**
srUpdateFunc **57**
srDeleteFunc **58**
srLockFunc **60**
srGetName **61**
srGetSize **62**
srGetMaxSize **63**
srHaveLock **64**
srNotify **65**
srPNotify **66**
srNumChannels **67**
srUserData **68**
srGetUserData **69**
srGetError **70**
srSearchLan **71**
srIsValid **72**

SR* srCreateServer

(int port, char* channels)

initialize streep in server-mode

Documentation

The srCreateServer function creates a server-side repository replication. Clients can connect to the server using the srCreateClient function. To create data in the repository, or to obtain the data that might be put there by a client, use srAllocMem, srAllocSerializedMem or srGetMem.

The function returns a handle that is used by many other functions to perform operations on the repository. Most such functions have the same semantics for both the server and the client. This means that once a handle is obtained, from either srCreateServer or srCreateClient, it can be used in the same manner regardless of weather it represents a client or a server.

The channels parameter is a string that describes a number of communication channels that are set up by the server. These channels are used to send and receive block-updates in cases where performance is more important than reliability and concurrency control. The string consists of a space separated list containing channel-types and and optional parameters like in the following example:

tcp udp mcast@255.1.1.1:1234 mcast@255.1.1.2:1234

In this example four channels would be set up, one using tcp (reliable), one using udp (unreliable) and two using multicast (unreliable). The tcp and udp channels do not need any parameters. The mcast channel need to be specified on the following format:

mcast@<multicast-address>:<port>[:ttl]

Where the address is a valid multicast address, i.e. in the range 225.0.0.0 - 239.255.255.255, and port is a valid udp port number. The optional ttl parameter is the time to live for the multicast packets, i.e. the maximum number of network hops that the packet will be routed over. If this parameter is omitted a default value of 16 will be used.

The channels are later referred to using integers. In the example above, the tcp channel would be referred to as channel 0, the udp channel as channel 1 and the two multicast channels as 2 and 3.

There is no restriction on the the number of channels, regardless of the type of the channel, that is possible to create.

How each channel should be used, i.e. what update that should be sent on which channel, is application dependent and is a choice for the application developer.

Parameters:

port - The port to listen to.
channels - A string describing the communication channels to be used.

Returns:

On success this function returns a pointer that represents the local repository replication. Otherwise, NULL is returned and the function srGetError can be used to retrieve information about the error.

See Also:

srCreateServer srClose

SR* srCreateClient

(char* host, int port)

initialize Streep in client-mode

Documentation

The srCreateClient function creates a client-side repository and attempts to connect it to a server. The local repository will initially be empty, even if there are data on the server. To create data in the repository, or to obtain the data that is already there, use srAllocMem, srAllocSerializedMem or srGetMem.

The function returns a handle that is used by many other functions to perform operations on the repository. Most such functions have the same semantics for both the server and the client. This means that once a handle is obtained, from either srCreateServer or srCreateClient, it can be used in the same manner regardless of weather it represents a client or a server.

Parameters:

host - The host to connect to.

port - The port that the host listens to.

Returns:

On success this function returns a pointer that represents the local repository replication. Otherwise, NULL is returned and the function srGetError can be used to retrieve information about the error.

See Also:

srCreateServer srClose

void srClose

(SR* sr)

dispose a repository replication

Documentation

The srClose function frees all memory and closes all network connections used by the repository sr. If the repository is a client, any locks owned by the client will be removed, and all shared blocks marked as volatile will be freed. If the repository is a server all clients will be disconnected. A client-side repository will not be useable once its server is closed.

Parameters:

sr - The repository that is to be disposed.

See Also:

srCreateServer srCreateClient

void* srAllocMem

(SR* sr, char* name, int size, int flags)

allocate a block of shared memory

Documentation

The srAllocMem function allocates a shared block of memory. The name parameter must be a unique name identifying the block. If a block with the specified name already exists, this function will fail. To get a reference to the block from another host, use the function srGetMem.

The size parameter specifies the maximum size of the data that are to be stored in the block, the actual size can be changed using srPNotify or srPUnlock.

The contents of the block will be sent as raw data, so beware of different byte orders on different hosts. To make sure that the data is correctly serialized, use the function srAllocSerializedMem.

The pointer returned by this function can be used in the same way as a pointer returned by standard allocation functions such as malloc. The pointer is also used to identify the block by other functions in the API.

Changes to a block made by one host need to be explicitly sent to other hosts. Two different mechanisms exist for this purpose; srLock/srUnLock for cases where concurrency control is important and srNotify for cases where performance is more important that concurrency control.

The flags parameter is formed by OR'ing one or more of the following values:

SR_VOLATILE - This flag requests that the block will automatically be freed when the client that created it disconnects from the server. The block will be freed regardless of weather the client disconnects gracefully or crashes.

SR_LOCK - This flag requests that the host that creates the block should initially hold a lock on the block. This is highly recommended in many cases. If this flag is not used it might be possible for another host to get the block before it has any contents (i.e. it has garbage) which most often will lead to undesirable results.

Parameters:

sr - The local repository replication.

name - The name used to refer to the block.

size - Maximum size of the block.

flags - Specifies how the block is to be allocated.

Returns:

On success this function returns a pointer that can be used to access the local copy of the shared block (in the same way as a call to malloc). On error, NULL is returned and the function srGetError can be used to retreive information about the cause of the error.

See Also:

srGetMem srAllocSerializedMem srUnLock srNotify

void* srAllocSerializedMem

(SR* sr, char* name, char* format, int flags)

allocate a block of serialized shared memory

Documentation

The srAllocSerializedMem funciton allocates a shared block of memory that will be automatically serialized and unserialized by the system. The name parameter must be a unique name identifying the block. If a block with the specified name already exists, this function will fail. To get a reference to the block from another host, use the function srGetMem.

The size of the block is given implicitly by the format parameter, and might be different on different hosts due to structure alignment issues. This size cannot be changed dynamically.

The format string describes the data that will be stored in the block. Such a descriptor string contains a sequence of the following characters:

```
c - char (8 bit)
s - short (16 bit)
l - long (32 bit)
f - float
d - double
```

Arrays can be defined by [n] where n is an integer. Structures can be encapsulated by { and }. As an illustration, consider the following structures:

```
struct a
{
  int x;
  char s[10];
};
struct b
{
  struct a x[10];
};
```

The struct a can be described by `"lc[10]"` and the struct b can be described by `"{lc[10]}[10]"`.

The pointer returned by this function can be used in the same way as a pointer returned by standard allocation functions such as malloc. The pointer is also used to identify the block by other functions in the API.

Changes to a block made by one host need to be explicitly sent to other hosts. Two different mechanisms exist for this purpose; srLock/srUnLock for cases where concurrency control is important and srNotify for cases where performance is more important that concurrency control.

The flags parameter is formed by OR'ing one or more of the following values:

SR_VOLATILE - This flag requests that the block will automatically be freed when the client that created it disconnects from the server. The block will be freed regardless of weather the client disconnects gracefully or crashes.

SR_LOCK - This flag requests that the host that creates the block should initially hold a lock on the block. This is highly recommended in many cases. If this flag is not used it might be possible for another host to get the block before it has any contents (i.e. it has garbage) which most often will lead to undesirable results.

Parameters:

sr - The local repository replication.

name - The name used to refer to the block.

format - A string specifying the format of the block.

flags - Specifies how the block is to be allocated.

Returns:

On success this function returns a pointer that can be used to access the local copy of the shared block (in the same way as a call to malloc). On error, NULL is returned and the function srGetError can be used to retreive information about the cause of the error.

See Also:

srGetMem srAllocSerializedMem srUnLock srNotify

void srFreeMem
(SR* sr, void* p)

free shared memory

Documentation

The srFreeMem function frees the shared memory block pointed to by p. This function is used to ultimately remove a shared block, to just remove a local copy use the function srReleaseMem instead.

The block will immediately be removed from the memory of the host that calls srFree. It will not be completely removed from the other hosts, however, before all hosts that have references to the block have given up those references by calling either srFreeMem or srReleaseMem.

When the first host calls srFreeMem the block will be invalidated, meaning that it is no longer possible to get references to the block using srGetMem, and that it is no longer possible to update or receive updates for the block. The block's name will also be removed from all internal tables, so that immediately following the call to srFreeMem it will be possible to create a new block with that name. The hosts that have references to the block, other than the one calling srFreeMem, will be notified of this invalidation event by a call to their deletion-callback, if they have registered such a callback for the block using the srDeleteFunc function.

Blocks that have been invalidated (i.e. freed by one host) but not yet freed by all other hosts can be seen as 'zoombie blocks', analogous to zombie processes on UNIX systems.

Parameters:

sr - The local repository replication.

p - The block to be freed.

See Also:

srAllocMem srReleaseMem srDeleteFunc

void* srGetMem

(SR* sr, char* name, int flags)

get a reference to a shared block

Documentation

The srGetMem function gets a reference to the shared block identified by name. The block must have been previously created using srAllocMem or srAllocSerializedMem, or else this function will fail.

It is not possible to get a reference to a block if another host currently holds a lock on that block. The default action in this case is for the system to block until that host has released the lock. This behavior can be changed using the flags parameter, see below.

The maximum size of the data that can be stored in the block is given when the block is allocated, either explicitly by the srAllocMem function or implicitly by the srAllocSerializedMem function. This size can be obtained using the srGetMaxSize function. The block's current size can be obtained using the srGetSize function.

The pointer returned by this function can be used in the same way as a pointer returned by standard allocation functions such as malloc. The pointer is also used to identify the block by other functions in the API.

Changes to a block made by one host need to be explicitly sent to other hosts. Two different mechanisms exist for this purpose; srLock/srUnLock for cases where concurrency control is important and srNotify for cases where performance is more important that concurrency control.

The flags parameter controls the action to be taken if a host holds a lock on this function and can be one of the following symbolic constants:

SR_NONE - The system will block until the host that holds the lock has released it. This is the default.

SR_NOBLOCK - This flag requests that the system should not block if a lock is held. In case there is a lock held, the contents of the block will be undefined (garbage) until the lock is released. When the lock is released the block will be filled with it's contents, and block's update-callback will be called, if such a callback is registered using the srUpdateFunc function.

SR_NOQUEUE - Same action as in the case of **SR_NOBLOCK** with the difference that the block will not be filled when the host that holds the lock releases it.

Since this function deals with locking in some sense, beware of deadlocks!

Parameters:

sr - The local repository replication.

name - The name of the block.

flags - Specifies how the block is to obtained.

Returns:

On success this function returns a pointer that can be used to access the local copy of the shared block (in the same way as a call to malloc). On error, NULL is returned and the function srGetError can be used to retrieve information about the cause of the error.

See Also:

srAllocMem srAllocSerializedMem srLock srUpdateFunc

int srIsServer

(SR* sr)

check repository type

Documentation

The srIsServer function checks weather the repository sr represents a server.

Parameters:

sr - The repository to be checked.

Returns:

If sr represents a server this function returns a positive value; if it represents a client this function returns 0.

See Also:

srCreateServer srCreateClient

void srReleaseMem

(SR* sr, void* p)

release shared memory

Documentation

The srReleaseMem removes the block indicated by p from the local repository. The block will not be removed from the server, meaning that other hosts can still use this block. To ultimately remove the block, use the function srFreeMem instead.

Parameters:

sr - The local repository replication.

p - The block to be released.

int srChannelSubscribe

(SR* sr, int channel)

subscribe to a communication channel

Documentation

The srChannelSubscribe function subscribes to the specified communication channel. This channel can be used to send block updates to other hosts using srNotify and to receive block updates from other hosts.

The channels are set up during the initialization of the server, see srCreateServer.

The number of available channels can be obtained using the srNumChannels function, and information about each channel can be obtained using the srChannelGetInfo function.

How each channel should be used, i.e. what update that should be sent on which channel, is application dependent and is a choice for the application developer.

Parameters:

sr - The local repository replication.

channel - The channel to subscribe to.

Returns:

On success this function returns a positive value. Otherwise, this function returns 0.

See Also:

srChannelUnSubscribe srChannelGetInfo srNumChannels srCreateServer

int srChannelUnSubscribe

(SR* sr, int channel)

unsubscribe from a communication channel

Documentation

The srUnSubscribe function unsubscribes from a communication channel. This channel can no longer be used to send block updates to other hosts or receive block updates from other hosts.

The channels are set up during the initialization of the server, see srCreateServer.

The number of available channels can be obtained using the srNumChannels function, and information about each channel can be obtained using the srChannelGetInfo function.

How each channel should be used, i.e. what update that should be sent on which channel, is application dependent and is a choice for the application developer.

Parameters:

sr - The local repository replication.

channel - The channel to unsubscribe from.

Returns:

On success this function returns a positive value. Otherwise, this function returns 0.

See Also:

srChannelSubscribe srChannelGetInfo srNumChannels srCreateServer

char* srChannelGetInfo

(SR* sr, int channel)

unsubscribe from a communication channel

Documentation

The srChannelGetInfo function returns a string containing a type and a possible options for the channel as specified when the channel was created by a call to the srCreateServer function.

The pointer returned points to memory allocated by the system and will remain valid until the repository is disposed using srClose. Do not attempt to free this memory.

Parameters:

sr - The local repository replication.

channel - The channel to retrieve information for.

Returns:

A string containing information about the channel.

See Also:

srChannelSubscribe srChannelUnSubscribe srNumChannels srCreateServer

void srProcess

(SR* sr, int waittimeout, int readtimeout)

process incomming / outgoing network traffic

Documentation

The srProcess function sends data that has been queued by the srNotify or srPNotify functions, for all channels that the host has subscribed to.

The srProcess also receives data from those channels, as well as data about other events. Depending on the data received, the callback functions registered using srUpdateFunc, srLockFunc or srDeleteFunc may be called.

The waittimeout parameter specifies the maximum number of milliseconds to spend waiting for data to arrive. If no data are received within that interval, this function will return without doing anything. If data are received, this function tries to receive all that data, but it does not spend more time than readtiemout millisec doing so.

The symbolic constants SR_NO_TIMEOUT and SR_NO_WAIT can also be used as values to the waittimeout and readtimeout parameters.

For the readtimeout parameter, however, it is not recommended to use these constants in most cases. A value of SR_NOWAIT will cause the system to process only the data currently in the network buffers, which might lead to more and more data that are 'queued up', which in turn leads to the data getting more and more outdated before it is received. A value of SR_NO_TIMEOUT might lead to the system getting stuck in a loop, doing nothing else than receiving data, if there are many fast hosts sending a lot of data on the network. The value that actually should be used for this parameter is application specific, but a value of 100 milliseconds might be a good value to try.

For the waittimeout parameter these constants might sometimes be useful. A value of SR_NOWAIT has the effect of polling the network for activity. Useful if you have a graphically intense application and don't want to spend to much time just waiting. A value of SR_NOTIMEOUT has the effect of waiting, in a system friendly manner, for data to arrive. Useful, for example in a server, if you have nothing to do unless you get some data.

If an error is detected this function returns immediately. The function srGetError can be used to get information about the cause of the error.

Parameters:

sr - The local repository replication.

waittimeout - The maximum number of milliseconds to spend waiting for activity.

readtimeout - The maximum number of milliseconds to spend receiving data.

int srLock

(SR* sr, void* p, int flags)

lock memory for mutual exclusion

Documentation

The srLock function attempts to lock a shared memory block for mutual exclusion. Note that the lock is *advisory*, it does not prevent any other host from actually writing to the memory, it just ensures that only one host has a lock.

If the block is already locked by another host, the default action for this function is to block until the lock is available. This behavior can be changed by passing one of the following symbolic constants as value for the flags parameter:

SR_NONE - The system will block until the host that holds the lock has released it. This is the default.

SR_NOBLOCK - This flag requests that the system should not block if a lock is held. In case there is a lock held, the srLock function will return 0, and the lock request will be queued. When the lock becomes available, the lock-callback will be called, if any such callback is registered using the srLockFunc function.

SR_NOQUEUE - Same action as in the case of **SR_NOBLOCK** with the difference that the lock request will not be queued; if the lock can not be obtained immediately it will not be obtained at all.

When a lock is obtained, using either of the mechanisms above, the contents of the block will also be updated in the local repository.

The functions srLock, srUnLock and srPUnLock represent one way of distributing updates to other hosts. These functions are intended to be used in cases where concurrency control is important. In the cases where concurrency control is not that important, the srNotify and srPNotify fnction provides a more efficient alternative.

Since this function deals with locking, beware of deadlocks!

Parameters:

sr - The local repository replication.

p - The block to attempt to lock.

flags - Specifies how the lock is to obtained.

Returns:

If a lock was obtained, this function returns a possitive value, otherwise 0 is returned.

See Also:

srPUnLock srUnLock srNotify srPNotify

int srPUnLock

(SR* sr, void* p, int size)

unlock memory and change size

Documentation

The srPUnLock function unlocks a shared memory block that has previously been locked using the srLock function. If the black was created using the srAllocMem, this function also changes the block's current size into the one specified by the size parameter. If the block was created using the srAllocSerializedMem, the bock's size will not be changed.

If the srPUnLock is called from a client, this function will send the block's contents to the server.

The functions srLock, srUnLock and srPUnLock represent one way of distributing updates to other hosts. These functions are intended to be used in cases where concurrency control is important. In the cases where concurrency control is not that important, the srNotify and srPNotify function provides a more efficient alternative.

Parameters:

sr - The local repository replication.

p - The block to unlock.

size - The new size of the block.

Returns:

If the block was successfully unlocked, this function returns a positive value. Otherwise, this function returns 0.

See Also:

srLock srUnLock srNotify srPNotify

int srUnLock

(SR* sr, void* p)

unlock memory

Documentation

The srPUnLock function unlocks a shared memory block that has previously been locked using the srLock function.

If the srUnLock is called from a client, this function will send the block's contents to the server.

The functions srLock, srUnLock and srPUnLock represent one way of distributing updates to other hosts. These functions are intended to be used in cases where concurrency control is important. In the cases where concurrency control is not that important, the srNotify and srPNotify function provides a more efficient alternative.

Parameters:

sr - The local repository replication.

p - The block to unlock.

Returns:

If the block was successfully unlocked, this function returns a positive value. Otherwise, this function returns 0.

See Also:

srPUnLock srLock srNotify srPNotify

void srUpdateFunc
(SR* sr, void* p, SR_UPDATEFUNC* func)

register update callback

Documentation

The srUpdateFunc registers a callback function to handle update events. The callback function will be called from within the srProcess function when the system receives block-updates from other hosts.

The callback function should be declared using the following prototype:

```
void func(SR *sr, void *p);
```

The function will be called from within srProcess with the sr parameter set to the local repository, and the p parameter set to the block that the callback is registered for.

It is possible to use the same callback function for several blocks. In those cases it might be useful to associate extra data with the block, i.e. data that are related to the block, but not distributed to other hosts. Such data can be registered using the srUserData function and retrieved using the srGetUserData function.

To remove a previously registered callback, call the srUpdateFunc with a NULL value for the func parameter.

Parameters:

sr - The local repository replication.

p - The block to register a callback function for.

func - The callback function.

See Also:

srDeleteFunc srLockFunc srProcess srUserData srGetUserData

void srDeleteFunc

(SR* sr, void* p, SR_DELETEFUNC* func)

register invalidation callback

Documentation

The srDeleteFunc registers a callback function to handle invalidation events. Invalidation events occur when a host frees shared memory using the srFreeMem function. The callback function will be called from within the srProcess function when the system receives such events from other hosts.

The callback function should be declared using the following prototype:

```
void func(SR *sr, void *p);
```

The function will be called from within srProcess with the sr parameter set to the local repository, and the p parameter set to the block that the callback is registered for.

It is possible to use the same callback function for several blocks. In those cases it might be useful to associate extra data with the block, i.e. data that are related to the block, but not distributed to other hosts. Such data can be registered using the srUserData function and retrieved using the srGetUserData function.

To remove a previously registered callback, call the srDeleteFunc with a NULL value for the func parameter.

In most cases the callback function should release the local references to the block. In its simplest form such a callback could look like this:

```
void deletecallback(SR *sr, void *p)
{
  srFreeMem(sr,p);
}
```

Not registering an invalidation-callback will often lead to memory leaks in the local repository.

Parameters:

sr - The local repository replication.

p - The block to register a callback function for.

func - The callback function.

See Also:

srFreeMem srUpdateFunc srLockFunc srProcess srUserData srGetUserData

void srLockFunc

(SR* sr, void* p, SR_DELETEFUNC* func)

register lock callback

Documentation

The srLockFunc registers a callback function to handle notification of lock-grants. The callback function will be called from within the srProcess function when the system receives lock-grants from other hosts. The callback will only be called for those locks that have been requested using SR_NOBLOCK flag for the srLock function.

The callback function should be declared using the following prototype:

```
void func(SR *sr, void *p);
```

The function will be called from within srProcess with the sr parameter set to the local repository, and the p parameter set to the block that the callback is registered for.

It is possible to use the same callback function for several blocks. In those cases it might be useful to associate extra data with the block, i.e. data that are related to the block, but not distributed to other hosts. Such data can be registered using the srUserData function and retrieved using the srGetUserData function.

To remove a previously registered callback, call the srLockFunc with a NULL value for the func parameter.

Parameters:

sr - The local repository replication.

p - The block to register a callback function for.

func - The callback function.

See Also:

srLock srDeleteFunc srUpdateFunc srProcess srUserData srGetUserData srHaveLock

char* srGetName

(SR* sr, void* p)

get name for a shared memory block

Documentation

The srGetName functions retrieves the name associated with a block. The block's name is the string passed to the srAllocMem or srAllocSerializedMem functions when the block was created.

It is not possible to get he name for a block that has been invalidated, i.e. freed by another host.

The pointer returned points to memory allocated by the system and will remain valid until the repository is disposed using srClose, or the block is removed using srFreeMem or srReleaseMem. Do not attempt to free this memory.

Parameters:

sr - The local repository replication.

Returns:

A string containing the name for the block.

See Also:

srAllocMem srAllocSerializedMem srFreeMem

int srGetSize

(SR* sr, void* p)

get the current size of a shared memory block

Documentation

The srGetSize function retrieves the current size of a shared memory block. If the block was allocated using srAllocMem this size may change during the lifetime of the block. In such case, to get the maximum size that this block can hold, use the srGetMaxSize function.

If the block was allocated using srAllocSerializedMem this size is static, but may be different for each host due to structure alignment issues.

Parameters:

sr - The local repository replication.

Returns:

The current size of the block.

See Also:

srGetMaxSize srAllocMem srAllocSerializedMem

int srGetMaxSize

(SR* sr, void* p)

get the maximum size of a shared memory block

Documentation

The srGetMaxSize function retrieves the maximum size of the data that the indicated block can hold. If the block was allocated using the srAllocMem function, the value returned by this function is always equal to the size passed to that function. If the block was allocated using the arAllocSerializedMem this value is given implicitly by the format string passed to that function.

Parameters:

sr - The local repository replication.

Returns:

The maximum size of the data that the block can hold.

See Also:

srGetSize srAllocMem srAllocSerializedMem

int srHaveLock

(SR* sr, void* p)

check lock status

Documentation

The srHaveLock function checks to see if the host that calls the function has a lock on the shared memory block indicated by the p parameter.

This function provides an alternative to using a lock callback-function when obtaining a lock asynchronously, i.e. when using the srLock function with the SR_NOBLOCK flag.

Parameters:

sr - The local repository replication.

Returns:

If the host that calls this function has a lock on the block p, this function returns a positive value. Otherwise, this function returns zero.

See Also:

srLock srUnLock srPUnLock srLockFunc

void srNotify

(SR* sr, void* p, int channel)

send block update

Documentation

The srNotify function schedules the data from the block p to be sent on the channel channel. The data will be sent during the next call to srProcess. The channel must have been previously subscribed to using srChannelSubscribe.

If several block updates are scheduled on the same channel before a call to srProcess, these updates will be sent in the same network message, which will usually be more efficient than sending them in individual messages. The updates will also be seen as atomic, i.e. they will all be processed at the same time at the host that receives them. If the channel is unreliable, these updates will either all reach a host, or none of them will.

Note that channels that are based on datagram services, such as udp and multicast channels, have a underlying maximum message size. This size is platform dependent, but typically around 64K. If this limit is reached no more data can be placed in the channel.

When the data arrive to a remote host, this host's update notification callback will be called, if the host has registered such a callback using the srUpdateFunc function. If the channel is a multicast channel, the local update callback will also be called during the next call to srProcess.

Parameters:

sr - The local repository replication.

p - The block that has been changed.

channel - The channel where to send the update.

Returns:

If the block was successfully placed in the channels output buffer, this function returns a positive value. Otherwise, 0 is returned.

See Also:

srPNotify srProcess srChannelSubscribe srChannelUnSubscribe

void srPNotify

(SR* sr, void* p, int channel, int size)

send block update and change size

Documentation

The srNotify function schedules the data from the block p to be sent on the channel channel. The data will be sent during the next call to srProcess. The channel must have been previously subscribed to using srChannelSubscribe.

If the block was allocated using srAllocMem, this function will also change the block's current size to the value of the size parameter.

If several block updates are scheduled on the same channel before a call to srProcess, these updates will be sent in the same network message, which will usually be more efficient than sending them in individual messages. The updates will also be seen as atomic, i.e. they will all be processed at the same time at the host that receives them. If the channel is unreliable, these updates will either all reach a host, or none of them will.

Note that channels that are based on datagram services, such as udp and multicast channels, have a underlying maximum message size. This size is platform dependent, but typically around 64K. If this limit is reached no more data can be placed in the channel.

When the data arrive to a remote host, this host's update notification callback will be called, if the host has registered such a callback using the srUpdateFunc function. If the channel is a multicast channel, the local update callback will also be called during the next call to srProcess.

Parameters:

sr - The local repository replication.

p - The block that has been changd.

channel - The channel where to send the update.

size - The block's new size.

See Also:

srNotify srProcess srChannelSubscribe srChannelUnSubscribe

int srNumChannels

(SR* sr)

get the number of available channels

Documentation

The srNumChannels function retrieves the number of communication channels available on the system.

Parameters:

sr - The local repository replication.

Returns:

The number of available channels.

See Also:

srCreateServer srChannelGetInfo

void srUserData

(SR* sr, void* p, void* data)

set per-block user data

Documentation

The srUserData function sets the user-data associated with a block. This data can later be obtained using the srGetUserData function.

The user-data exist to associate local data with a shared block. There is no way for one host to obtain the user-data associated with a block by another host.

Parameters:

sr - The local repository replication.

p - The shared memory block.

data - The data to associate with the block.

See Also:

srGetUserData srAllocMem srAllocSerializedMem srUpdateFunc srDeleteFunc srLockFunc

void* srGetUserData

(SR* sr, void* p)

get per-block user data

Documentation

The srGetUserData function retrieves the user-data associated with a block. This data should have been registered using the srUserData function.

The user-data exist to associate local data with a shared block. There is no way for one host to obtain the user-data associated with a block by another host.

Parameters:

sr - The local repository replication.

p - The block to retreive the user-data for.

Returns:

The data associated with the block.

See Also:

srUserData srAllocMem srAllocSerializedMem srUpdateFunc sDeleteFunc srLockFunc

int srGetError

(void)

get error information

Documentation

The srGetError function returns the error code for the last occurred error. If no errors have been detected since the last call to srGetError, this function returns SR_NO_ERROR. If an error is detected, the srGetError returns a value equal to one of the following symbolic constants:

SR_NO_ERROR - No error has been detected.

SR_INVALID_VALUE - A numeric argument is out of range or otherwise invalid in the current state.

SR_INVALID_OPERATION -The specified operation is not allowed in the current state.

SR_OUT_OF_MEMORY - There is not enough memory left to execute the function. This error can occur both on the server, if it runs out of memory, or on the client, if either the server or the client runs out of memory.

SR_NO_CONNECTION - The client has lost its connection to the server. After this error has been detected, no more data can either be sent or received. This error can only occur on a client. If a server detects a lost connection to one of its clients, this is treated as if that client has disconnected, and no error is reported.

Returns:

The current error status, or SR_NO_ERROR if there is no error.

char* srSearchLan

(int port, int retries, int timeout, char* addr)

search local area network

Documentation

The srSearchLan function searches the local area network for a server running on the specified port. The function does this by sending a broadcast and waits for a response from a server. If no response is received, this function waits for the specified number of milliseconds and sends another broadcast. This procedure is repeated as many times as the value of the retries parameter. The maximum possible waiting time is retries * timeout millisec.

Returns:

On success this function returns a pointer to the destination address addr. On error this function returns NULL.

int srIsValid

(SR* sr, void* p)

check block validity

Documentation

The srHaveLock function checks to see if a block is valid.

This function provides an alternative to using a deletion callback-function to detect blocks that have been remotely invalidated.

Invalidation occur when a host frees shared memory using the srFreeMem function. If a block has been invalidated, it is no longer possible to get references to the block using srGetMem, and that it is no longer possible to update or receive updates for the block. The block's name will also be removed from all internal tables, so that immediately following the call to srFreeMem it will be possible to create a new block with that name.

Parameters:

sr - The local repository replication.

p - The block to check for validity.

Returns:

If the block is valid this function returns a positive value. Otherwise, this function returns zero.

See Also:

srFree srDeleteFunc

References

[1] Birell, A. D. and Nelson, B. J. Implementing Remote Procedure Calls. *ACM Trans. On Computer Systems*, vol. 2, pp. 39-59, Feb 1984.

[2] Olson, M. Introduction to Corba. *LinuxWorld*. Sep 1999. Retrieved Mar 14, 2002 from the World Wide Web: *http://www.linuxworld.com/linuxworld/lw-1999-09/lw-09-corba_1.html*

[3] Tanenbaum, A. S. Distributed operating systems. Prentice-Hall, Inc., 1995. ISBN 0-13-143934-0

[4] Einstein, A. Zur Elektrodynamik bewegter Körper. *Annalen der Physik*. Verlag von Johann Ambrosius Barth, Leipzig 1905.

[5] Carreiro, N. and Gelernter, D. The S/Net's Linda Kernel. *ACM Trans. On Computer Systems*, vol. 4, pp. 110-129, May 1986.

[6] Singhal, S. and Zyda, M. Networked Virtual Environemnts, ACM Press 1999. ISBN 0-201-32557-8

[7] Pope, A. The SIMNET network and protocols.*Technical Report 7102*. Cambridge, MA: BBN Systems and Technologies, July 1989.

[8] Carlsson, C. and Hagsand, O. DIVE – a Multi-User Virtual Reality System. *IEEE VRAIS*, Sept 1993.

[9] Java™ Shared Data Toolkit Home Page, Sun Microsystems 1995-2002. Retreived Mar 12, 2002 from the World Wide Web: *http://java.sun.com/products/java-media/jsdt/*

[10] Leigh, J. CAVERNSoft G2, University of Illinois at Chicago. Retrieved Mar 12, 2002 from the World Wide Web: *http://www.openchannelsoftware.org/projects/CAVERNsoft_G2/*

[11] DIVERSE Toolkit, Virginia Polytechnic Institute and State University. Retreived Mar 12 from the World Wide Web: *http://www.diverse.vt.edu/DTK/*

[12] Brutzman, D., Zyda M., Watsen, K. and Macedonia M. Virtual Reality Transfer Protocol (vrtp) Design Rationale. *WET ICE: Sharing a Distributed Virtual Reality.* Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1997.

[13] MacIntyre, B. and Feiner, S. A Distributed 3D Graphics Library. Columbia University.

[14] Anupam, V. and Bajaj, C. L. Shastra: Multimedia Collaborative Design Environment. *IEEE Multimedia 1(2), Summer 1994.*

[15] Bernier, Y. W. Half-Life and Team Fortress Networking. *Game Developer's Conference proceedings, 2000*.

[16] Datareel Home Page. glNET Software, 2001. Retrieved Aug 14, 2001 from the World Wide Web: *http://glnetsoftware.com/datareel*

[17] RFC 1014: XDR: External Data Representation, Sun Microsystems, Inc. June 1987.

[18] IBM Pitsburg Lab Documentation, IBM Corporation, 2001. Retrieved Aug 15, 2001 from the World Wide Web: *http://www.transarc.ibm.com/Library/documentation/index.html*

[19] Moss, J. E. B. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering, 18(8):657--673*, August 1992. Retrieved Aug 15, 2001 from the World Wide Web: *http://citeseer.nj.nec.com/moss92working.html*

[20] InterAct: Virtual Sharing for Interactive Client-Server Applications, S. Parthasarathy and S. Dwarkadas, Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, May 1998. Retrieved Sep 19, 2000 from the World Wide Web: *http://www.cs.rochester.edu/u/sandhya/papers/lcr98_interact.ps*

[21] Standard Template Library Programmer's Guide. Silicon Graphics, Inc. © 1993-2002. Retrieved Apr 2, 2002 form the World Wide Web: *http://www.sgi.com/tech/stl/*

[22] Tanenbaum, A. S. Computer Networks. Prentice Hall PTR 1996. ISBN 0133499456.

[23] Seipel S. and Lindkvist M. Interactive Graphical Environments for Collaborative Learning and Teaching. Uppsala University Research Report 2001:9, ISSN 1403-7572.

[24] Wallenberg Global Learning Network. Retrieved Apr 2 form the World Wide Web: *http://www.wgln.org*

[25] Command & Control, Swedish National Defence College, Department of Operational Studies, 2002. Retrieved Mar 11, 2002 from the World Wide Web: *http://www.militaryscience.org*

[26] Seipel S. and Lindkvist M. Project Aqua progress report, 2001. Department of Information Science, Uppsala University. Order 1853-010614-OpI

[27] Olsson, E. Safer navigation at sea through augmented reality? Uppsala University Technical Report. ISSN 1404-3203.

[28] Aronson, J. Dead Reckoning: Latency Hiding for Networked Games. Gamasutra. 2000. Retrieved Sep 27, 2000 from the World Wide Web: http://www.gamasutra.com/features/special/online_report/dead_reckoning.htm

[29] Mills, D. Time Synchronization Server. 2002. Retrieved Mar 8, 2002 from the Word Wide Web: *http://www.eecis.udel.edu/~ntp/*

[30] The Open Source Toolkit for SSL/TSL, The OpenSSL project, 1999. Retrieved Mar 8,2002 from the World Wide Web: *http://www.openssl.org/*

[31] ITU-T RECOMENTATION T.122, International Telecommunication Union, February 1998