Embedded Distributed Systems

Distributed Processing in a Box

Dominic Herity
Dominic.Herity@redplain.com
Red Plain Technology
www.redplain.com

Distributed Processing is used increasingly in Embedded Systems. As processors become smaller and cheaper, more of them are used in more products. The Software Architect is asked to design solutions that make effective use of additional processors, but without sacrificing performance and without needing extensive re-engineering when hardware changes. This workshop describes the applicability of traditional distributed processing to meeting these challenges and goes on to explore the unique attributes of embedded distributed systems. It uses a real world example to explore the issues that arise and suggests guidelines on designing embedded distributed solutions that are efficient and reusable in different hardware environments.

CONTENTS

Route Table Maintenance....5

Simple Router Architecture....5

High Performance Router Architecture....6

Software Complications....8

A Distributed System....9

- Weak and Strong Messaging Guarantees....10
- Weak Guarantee: Asynchronous, Unreliable, Unordered....10
- Strong Guarantee: Synchronous, Reliable, Ordered....11
- Message Passing Service – Sockets....12

Router Example Using Message Passing....12

Remote Procedure Call....13

Distributed Objects....16

- Distributed Virtual Threads....19

Benefits of a Distributed Object Model....19

- Remote State Encapsulation....19
- Optimized Local Invocations....19
- Decoupling of Application from Hardware Architecture....20
- Distributed Application Portability....20
- Simplified Handling of Computation on Many Processors....20
- Exception Safety - A Free Lunch?....22

Distributed Object Technologies....23

- CORBA....23
- High Performance Distributed Objects....24

Latency..25

Bandwidth..26

Processor Speed to Latency Ratio..26

Reliable Physical Layer..26

Physical Security...26

Number of Processors, Concurrent Startup and Shutdown............27

INTRODUCTION

A “distributed system in a box” is shorthand for a system which has several independent processors connected by a bus or other communication mechanism.. In this paper, we will discuss the increasing relevance of distributed systems methodology and technology to embedded system design as more and more products are designed with multiple processors inside, becoming, in effect, distributed systems in boxes. We will use a particular aspect of IP router software as an example. We will apply different distributed system paradigms to the example and examine the consequences. The paradigms discussed are:

- message passing
- remote procedure call
- distributed objects.

We will explore the costs and benefits that come with the higher levels of abstraction.

Specific technologies discussed are Berkeley Sockets, RPC, CORBA and Red Plain’s distributed object technology, redFOX.

We will then consider how the traditional distributed system model maps to a distributed system in a box. We will point out areas where invalid assumptions may lead to poor results.

We finish with a summary of characteristics that may differ between a traditional distributed system and a distributed system in a box.

ROUTER EXAMPLE

Consider the design of a typical embedded system which has evolved over the years. We’ll take an IPv4 router as an example. I choose this example because

- it has well defined core functionality
- it has been around for a (relatively) long time
- its hardware architecture has changed radically in that time
- these changes are typical of changes in embedded systems hardware architecture
- the challenges for the software developer and the solutions are also typical of embedded system designs.

An IP router is a device that has many network ports, each connected to a different network. When it receives an IP packet on a port, it reads its destination IP address and uses it to look up a route table to find its next hop IP address and other information. The next hop IP address is the IP address to which the router must send

the packet. The next hop address tells implicitly which network and therefore which port the packet is to be sent out on.

The route table is central to the operation of the router. It is a collection of key/value pairs. A key consists of a prefix IP address and a bit mask indicating how many bits of the prefix are significant. The value consists of a next hop IP address and other information needed to process the packet. Route lookup consists of finding a prefix in the route table that matches a packet's destination IP address. If multiple prefixes match, the prefix with the most significant bits is used. Next hop address lookup is specified in [] section 5.2.4.3.

Route Table Maintenance

The route table is maintained by a routing protocol, which means that it changes while the router is running. This routing protocol talks to its peers on neighboring routers and alters the routing table piecemeal as a result of its information received. Route table entries are added and deleted on the fly.

In C, the API to the route table might look something like Figure 1.

```
typedef uint32_t ip_address;

int route_table_add(struct route_table* table, ip_address prefix,
                    ip_address mask, ip_address next_hop);
                    /* Add entry for prefix, mask to table */

int route_table_delete(struct route_table* table, ip_address prefix,
                    ip_address mask);
                    /* Delete entry for prefix, mask from table */
```

Figure 1: Route Table API

The parameter `table` identifies which route table is referenced.[1] The parameters `prefix` and `mask` indicate the key to be added to or deleted from the route table. The parameter `next_hop` is the next hop address value for the corresponding key. The return value indicates success or failure.

Simple Router Architecture

The most straightforward way to build a router is to take a general purpose computer like a PC with a processor, a bus and a memory, and then add network cards and software. In this hardware architecture, a network card transfers a packet arriving at an ingress port to a memory buffer using Direct Memory Access (DMA). It

[1] A router may use more than one route table, for example, when implementing Virtual Private Networks (VPNs).

then adds it to a software queue for processing and interrupts the processor. Software takes packets from the queue, examines each packet, performs the route lookup, then puts the outbound packet on a queue for transmission by the egress port network card.

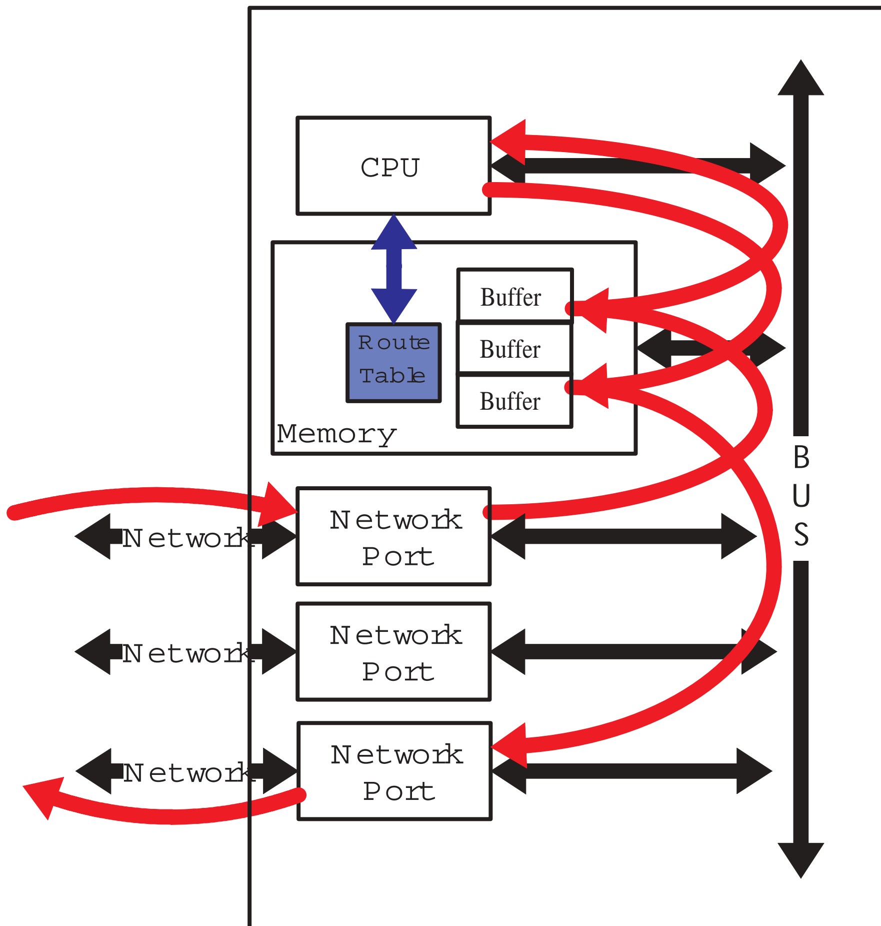

Figure 2: Simple Router Architecture

Figure 2 illustrates this process. Black arrows represent data paths. Red arrows represent a packet arriving, being processed and leaving. The blue arrow represents route table lookup and maintenance by the processor.

High Performance Router Architecture

The simple router architecture doesn't scale very well. Every packet from every port is written to the same memory, where it is read and written by the same processor and read by the egress port. The bus connecting the processor, the memory and the network cards is a bottleneck.

In modern high performance routers, the bus is replaced by a switch fabric. A switch fabric is an NxN matrix of connections. It has N inputs and N outputs and it can

be programmed to connect each output to any input. The advantage of a switch fabric is that it has a much higher aggregate bandwidth than a bus. Whereas a bus can receive or transmit one packet at a time, an NxN switch fabric can transfer up to N packets from ingress to egress simultaneously.

But for this to work, the central processor must be taken out of packet forwarding. Route lookup must be done at the ingress port on each network card. Each network card must have a processor and a replica of the route table.[2] The route table protocol runs on the central processor and distributes updates to the replicas.

This is illustrated in Figure 3. An arriving packet (red arrows) arrives at an ingress port. Route table lookup is done on the network card and the packet is sent via the switch fabric to a chosen egress port. The central processor maintains a master route table and notifies changes to the network card processors, so that each has an up to date replica of the master route table.

[2] The port hardware may use a combination of hardware and software to do packet processing. For simplicity, we will just consider software processing.

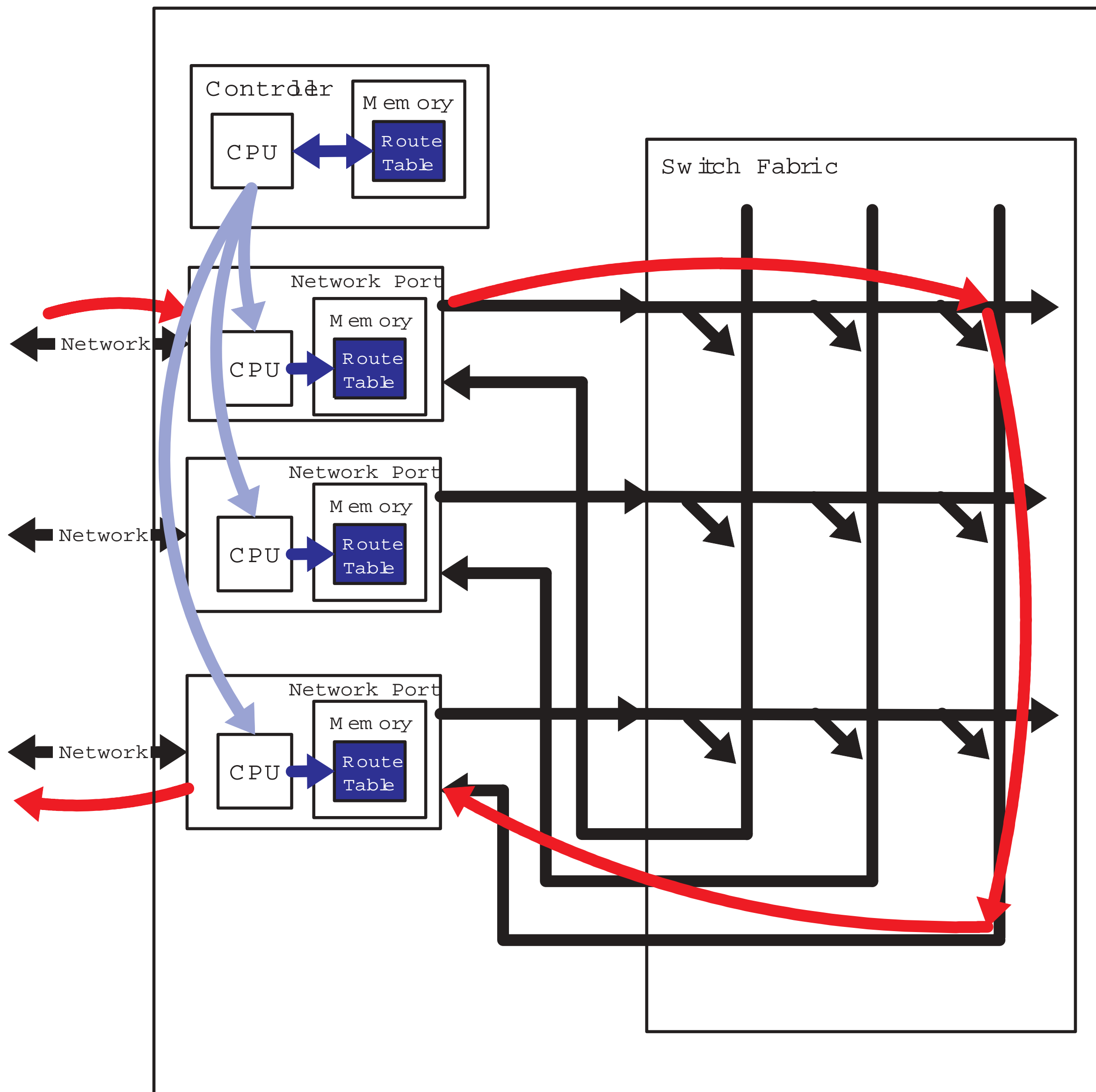

Figure 3: High performance Router Architecture

Software Complications

The high performance router is a much more challenging environment for the software architect. In the simple router, all the software ran on a single processor, one thread at a time, and all the system state was directly accessible in the same memory.

In the high performance router, the majority of packets aren't even seen by the central processor. There are as many threads running at a time as there are processors and a thread on a given processor may have incomplete or out-of-date information about the state of the system as a whole. Different processors have different, potentially contradictory, views of the system state.

The route table has to be replicated at each port and we must arrange for routing protocol packets and other local traffic to be transferred between ports and the central processor. We need to worry about keeping the replicated route tables consistent and about the ordering of route table updates.

For example, when a route table entry is added, the processor on the egress port for that route must be told before the other ports, or the other ports may send it packets that it doesn't know what to do with.

Other router functionality, such as access control and statistics gathering, are also more complex in the high performance router.

A Distributed System

Fortunately, the problem of performing computation on multiple collaborating processors has been studied for many years. Coulouris, Dollimore and Kindberg [] define a distributed system as "*one in which components located at networked computers communicate and coordinate their action only by passing messages*". This description certainly applies to our High Performance Router.

In [] pp 5-7, Mullender says that the possibility of failures in communication and processors is important. This is not always an issue in systems where processors share a bus and a power supply, but it sometimes is an issue. Fault tolerant routers are now being built with exactly this property to provide high availability. Continuing service in the event of processor failure affects many aspects of application design. So there is a lot to be gained by thinking of our High Performance Router as a distributed system.

VARIATIONS ON MESSAGE PASSING

Message passing seems like a straightforward proposition. A client encodes a message and sends it to a server.[3] The server decodes it and takes some action that depends on the content of the message. Part of the action may be to encode and send a response back to the client. Physically, the message may be passed though a variety of communication mechanisms, including shared memory, LANs, the Internet and RS232 ports.

But passing message among multiple processors raises a number of questions. Does the sender wait for a response or run ahead? If it runs ahead, how can it collect

[3] The terms client and server are sometimes used to mean slightly different things. In this paper, we refer to client and server in the context of passing an individual message and getting an individual response. A client sends a message to a server and the server responds to the client. The roles are transient rather than fixed. It would not be unusual, for example, for a server to send a message to another entity, becoming a client in turn, or to send a message back to the client, in which case roles are temporarily reversed.

a response? If it doesn't run ahead, what does it do while it's waiting? What guarantees are given about the ordering of messages? Without a response, how does the client know when messages are processed (as opposed to received)? Are messages even guaranteed to be delivered?

Weak and Strong Messaging Guarantees

Let us consider doing route table maintenance using two different message passing services and explore the consequences. The differences are extreme for the purposes of exposition and real world message passing services fall somewhere between these two extremes and make different tradeoffs between flexibility and ease of use.

Both services have an API with two functions. The client constructs a message in a buffer and calls a 'send' function with the buffer as a parameter. The server calls a 'receive' function which blocks until an incoming message is available. The 'receive' function then returns a buffer containing a copy of the message sent by the client. For route table maintenance, the message passed consists of an 'add' or a 'delete' command with appropriate parameters.

Weak Guarantee: Asynchronous, Unreliable, Unordered

The two message passing services are different, though, in the semantics of the service they provide. The first service provides weak guarantees and is rather like UDP/IP. The send function returns as soon as possible, before the message is delivered. It minimizes copying by accessing the buffer after the send function returns. It allows the client thread to 'fire and forget' multiple messages, maximizing throughput if message delivery is slow. On the server side, it delivers a sequence of messages for processing. Client and server are decoupled and both can keep busy without waiting for the other. But the service does not guarantee to deliver the message. Nor does it guarantee to deliver messages in the order that they are sent. In short, it provides asynchronous, unordered and unreliable message delivery. It also returns promptly, but may read the buffer after return to complete message transmission. These semantics unfortunately cause problems.

First, if a message requires a response, how do we deliver the response and associate it with the message that it refers to? Route table adds and deletes may succeed or fail and the result must be conveyed to the client.

Second, how does the client know when the operation is complete? If it needs to add a route table entry at the specified egress port before other ports, it must wait for the addition to be complete at the route's egress port before adding it to the route tables at other ports. So we can see that a message may require a response even if the only information in the response is the implicit information that the message has been processed. Note that this is different from knowing that the message has been delivered.

Third, what happens if a message is lost? Correct operation of our router requires that copies of the route table at the ports match the master copy in the controller, so we must ensure that lost messages are re-transmitted.

Fourth, what if a message is delivered out of sequence? If two messages are sent, the first to delete an entry with a given key and the second to add an entry with the same key, the affected route table should finish up with the new entry. If the messages are processed out of order, the entry finishes up deleted instead. So we must add sequence numbers to the messages and ensure they are processed in order.

Finally, when can the client re-use the buffer with the message? Because the service may access the buffer after the send function has returned, the message passing service has effectively acquired ownership of the buffer. This ownership must somehow be relinquished when message transmission has finished

So an asynchronous, unreliable, unordered messaging service with transfer of buffer ownership has significant hidden costs that can rarely be avoided. These hidden costs are usually not obvious during software development. It may be necessary to test software using a deliberately badly behaved version of the service that drops, shuffles and stalls messages at high rates to make sure that the software can handle these eventualities in the field.

Strong Guarantee: Synchronous, Reliable, Ordered

Now consider a message passing service with stronger guarantees. A reliable, synchronous, ordered service sends messages reliably in order. The client side 'send' function (optionally) blocks until a response is received and it returns the response to another client-supplied buffer. The server side 'receive' function provides a buffer for the server to place its response. So the client is blocked until the message processed and a response received. After 'send' returns, the client knows that the message has been delivered, processed and answered or that an error has occurred. It isn't concerned about buffer ownership. Because messages to add and delete routes are processed in order, the state of the remote route table is known. If an add or a delete fails, the failure can be handled without the complication of outstanding messages.

The price for this simplicity is that the client is blocked until the operation is complete and a distributed system where processors spend a lot of their time idle is not very useful. For the client processor to get useful work done while waiting for a response, we need a multi-threaded operating system

For most applications, the extra complexity of reliable ordered message delivery and a multi-threaded operating system is well paid for in application simplicity and correctness. The higher throughput of asynchronous messaging can be achieved using multiple threads. Messaging with weak guarantees is usually favored for performance reasons, but we need to make sure that like is being compared with like.

If the application is forced to ‘strengthen’ the messaging service, the result may be more expensive, less flexible and poorer in performance than adopting a well designed strong messaging service to start with.

We see below that messaging services with strong guarantees provided by hardware are common in embedded systems

MESSAGES, PROCEDURES AND OBJECTS

There are abstractions that we can put on top of message passing to make the design and implementation of distributed applications easier. These abstractions increase application flexibility and allow some of the software that is needed for distributed processing to be automatically generated. To see the purpose and value of these abstractions, we will explore the implementation of distributed route table updates using three abstractions – message passing, remote procedure calls and distributed objects.

Message Passing Service – Sockets

Perhaps the best known message passing API is Berkeley Sockets which are provided as part of UNIX, Windows and other operating systems. They allow a client and a server to set up two-way connection using Internet Protocol or other network protocols. Message delivery may be ordered and reliable or unordered and unreliable. Two of the main function calls in the sockets API in Linux are:

```
int send(int s, const void *msg, size_t len, int flags);
int recv(int s, void *buf, size_t len, int flags);
```

where s is the socket number, msg and buf are message and buffer pointers, len is message length or buffer length and flags is a set of options. Both functions normally return the number of bytes sent or received.

Other function calls are used to set up, configure and tear down a connection. See Linux man pages for more information.

Router Example Using Message Passing

It is instructive to list what we have to do to keep route tables replicas up to date in a distributed router using sockets. When the routing protocol updates the route table on the central processor, a message must be sent to each network card processor instructing it to make a corresponding update to its own route table. The network card processor must then return an error code to the central processor. To make all this work, we have to:

1. Run a thread on each network card processor to wait for route table messages and act on them
2. Set up a socket connection between the correct threads on the central processor and on each network card processor.
3. Devise a set of messages and replies to be exchanged between the central processor and the network card processors
4. Arrange to send a message to the correct thread on each network card and get each reply (central processor).
5. Construct and send each message type (central processor),
6. Receive and interpret message (network card processor),
7. Dispatch message to appropriate handler function (network card processor, route_add, route_delete, ...)
8. Act on message (network card processor)
9. Construct and send a reply (network card processor)
10. Receive and interpret the reply (central processor)

When heterogeneous processors exchange messages, we must ensure that data representations are agreed and understood. If, for example, we have a little-endian processor and a big-endian processor exchanging messages and responses in binary, one or the other must shuffle transferred data. A less obvious data representation issue arises when source code is compiled by different compilers. Different C/C++ compilers may lay out a struct differently, causing problems when structs are passed in messages.

Of the steps listed above, only steps 4 and 8 are specifically concerned with route table updates. The rest can be termed *middleware*, software in the middle that connects things up. Such middleware can be generated automatically from an interface specification written in an Interface Definition Language (IDL).

Remote Procedure Call

A Remote Procedure Call (RPC) service uses an underlying messaging service to make user-supplied procedures (functions in C terminology) remotely callable. It uses a user-supplied interface specification to generate the type of middleware described above. This is represented graphically in Figure 4.

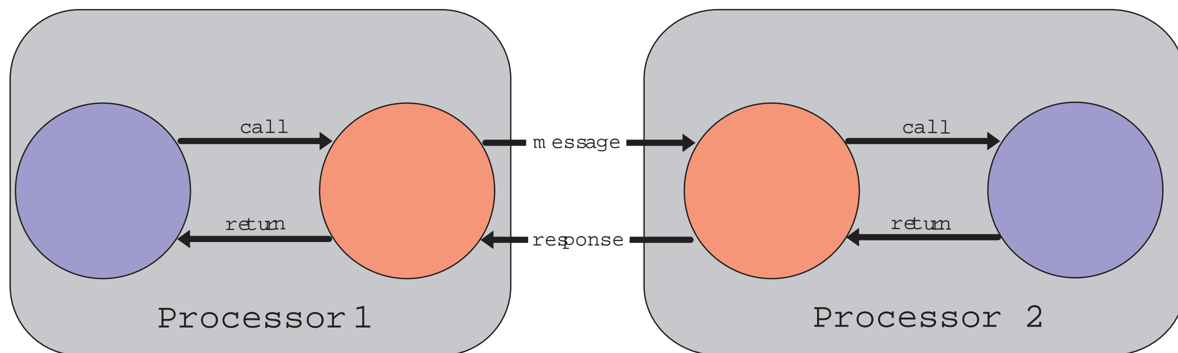

Figure 4 : Remote Procedure Call

The user writes an interface definition file in a C-like syntax called Interface Definition Language (IDL). This file is read by the program rpcgen, which generates a set of C files:

- A server stub C file that sets up sockets, receives messages, parses them, converts parameters to the correct representation, calls the appropriate user-supplied function, gathers the returned information, constructs a reply message with the appropriate data representation, sends a reply message then continues to receive messages.
- A client stub C file with a function matching each user-supplied function in the server. This function has the same parameters and return values as the user-supplied function. It converts data representation and marshals the converted parameters into a message for the server stub. It sends the message, waits for the reply, converts the representation of the data in the reply and returns the return value.
- A C file containing functions to convert the representation of all parameter types used in all remotely accessible functions.
- A header file defining constants, message and reply structures used by the generated C files.

The RPC version standardized by the Internet Engineering Task Force (IETF) is described in [] and the Distributed Computing Environment (DCE) specification is described in [].

```
#include "route_table_info.h" /* route_table_info */

typedef uint32_t ip_address;

program route_table {
    version VER {
        int route_table_add(route_table_info, ip_address, ip_address,
                            ip_address)=1;
```

```
        int route_table_delete(route_table_info, ip_address,
                               ip_address)=2;
    }=1;
}=10001;
```

Figure 5: RPC Interface Definition for Route Table

Figure 5 shows an RPC-style interface specification for a route table. The developer must provide the functions

```
int* route_table_add_1_svc(route_table_add_in *, struct svc_req *);
int* route_table_delete_1_svc(route_table_delete_in *,
                              struct svc_req *);
```

and link them with the server. To remotely call these functions, the client code must call

```
int * route_table_add_1(route_table_add_in *, CLIENT *);
int * route_table_delete_1(route_table_delete_in *, CLIENT *);
```

From the 11 lines in Figure 5, rpcgen generates 290 lines of C, which is a considerable saving in labor compared to using message passing directly. igure 6 shows diagrammatically how rpcgen is used to generate RPCs.

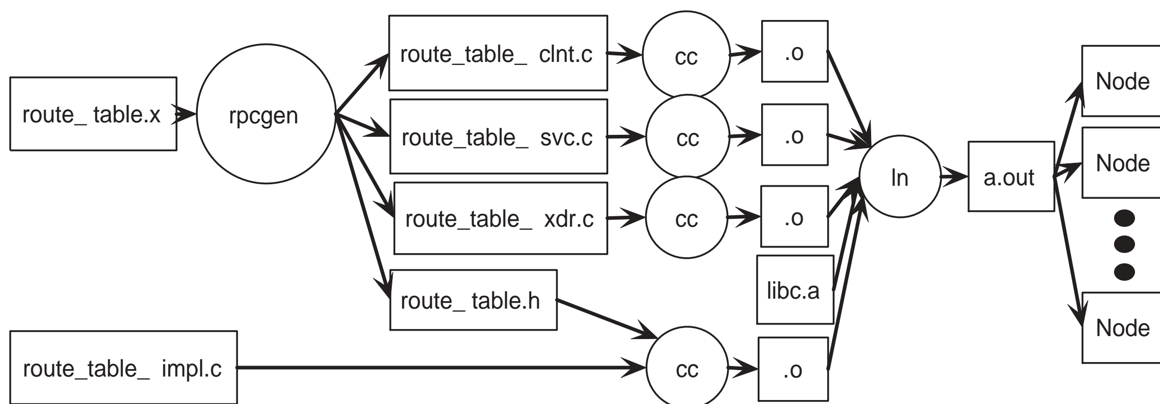

Figure 6: Generating Middleware with rpcgen

Despite appearances, RPC isn't quite like an ordinary function call. One important difference is that the type of information that the parameters carry is restricted. It doesn't make sense to pass a pointer to private memory to another processor, because the other processor can't access the data pointed to. Simple non-pointer data types can be passed and returned, as can structures containing only non-pointers. An RPC service may interpret a pointer parameter to mean that data pointed to is to be copied to the server, possibly modified and copied back. This tactic gives some of the semantics of a C pointer parameter, although only the value pointed to (and not adjacent values in an array) may be modified. But we must remember that passing a pointer in C is inexpensive, but copying a value to and from an RPC server is not.

Another important difference between RPC and ordinary function calls is the possibility of processor failure. C functions may fail to do what is expected of them, but they never actually fail to run. RPC calls may fail either because of a processor failure or communication failure and this possibility needs to be dealt with after each RPC call in the client code.

To update route tables on multiple processors, we must run an RPC server on each processor and call the RPC function on all these servers. It's a bit cumbersome to have an RPC process for something as simple as a route table, so in reality, the route table API would be added into an RPC server with other functions.

RPC is easier to use than message passing because it has a higher level of abstraction. It hides the details of message passing under a familiar function call paradigm. It frees us from writing of the order of 100 lines of code per called function.

Distributed Objects

It is natural to model a route table as an object, comprising both the table data itself and the procedures (methods) for adjusting it.. The API in Figure 1 can be viewed as an object interface, if we consider the route_table parameter as a reference to an object with methods route_table_add and route_table_delete. If we had written the API in C++, we might have come up with something like Figure 7.

```
typedef  uint32_t ip_address;

class route_table {
public:
    bool add(ip_address prefix, ip_address mask, ip_address next_hop);
    // Add entry for prefix, mask to table

    bool delete(ip_address prefix, ip_address mask);
    // Delete entry for prefix, mask from table
    // ...
};
```

Figure 7: C++ Route Table API

Just as local function calls can be generalized to allow calls between threads on separate processors using the RPC mechanism, so the standard object model can be generalized to a *distributed* object model.

In a distributed object model, objects may be located on different processors. A remote object reference (ROR)[4] fulfills the same role in a distributed application as a reference does in a non-distributed application. An ROR includes the information

[4] Different terms are used in different technologies to represent the concept of a remote object reference. The term ROR is chosen in this paper because it is not technology specific.

needed to get to the processor and find the object. It also has a method for each method of the target object's class (see Figure 8).

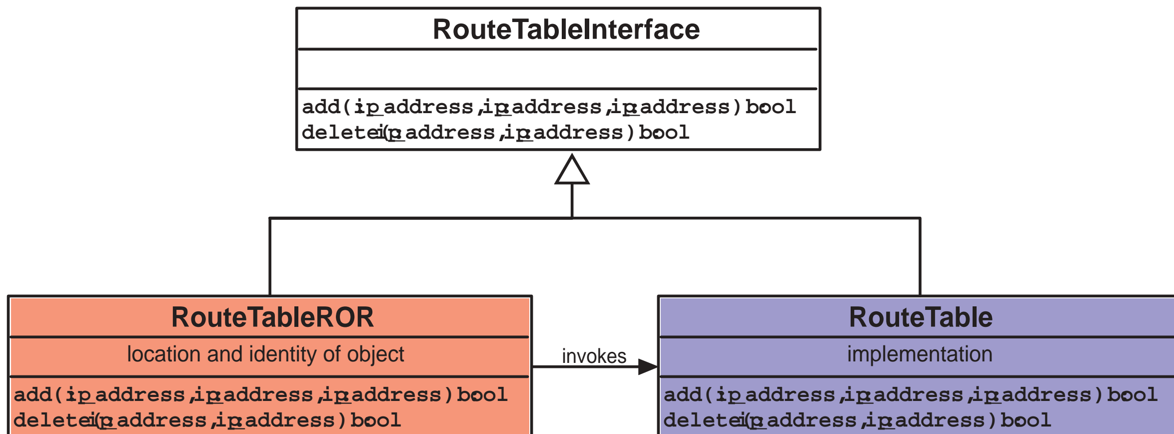

Figure 8: Remote Object Reference and Target Object

Each ROR method calls the corresponding method on the target object using a message passing infrastructure. When a method is invoked on an ROR, the middleware determines the location of the object and an object pointer from the ROR and otherwise behaves pretty much like RPC. Parameters are marshaled, messages sent, replies received and values returned.

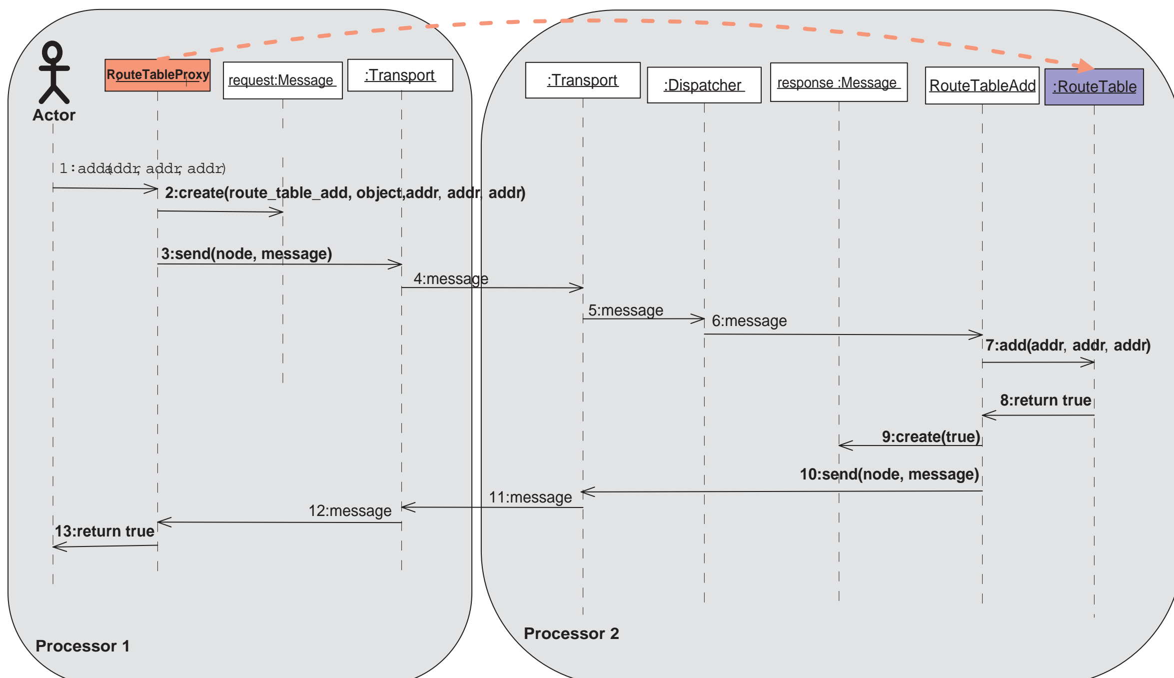

Figure 9: Remote Object Invocation

The figure above shows a remote invocation on a route table object in the form of a message sequence chart.

1. Client calls add method on a Route Table ROR.
2. The ROR determines that the object is remote and constructs a message containing the method identifier, object identifier and parameters.
3. The ROR calls the underlying message transport to deliver the message to the processor containing the target object.
4. The message is transmitted to the server processor.
5. The message is delivered to a message dispatcher responsible for object invocations.
6. The dispatcher selects a server thread to run the invocation and causes this thread to call the appropriate function to decode the message.
7. The message is decoded, the parameters and object pointer are reconstructed. The method on the target object is called.
8. The called method returns its result.
9. The result is encoded into a response message.
10. The message transport is called to send the response.
11. The response is delivered to the client processor.
12. The response message is delivered to the ROR.
13. The ROR decodes the response and returns it.

In object oriented programming, it is a good discipline to only access objects by calling their methods. In distributed object programming, this is a physical necessity because the object and its member variables may be on a different processor.

An ROR is a straightforward generalization of an object reference in C++ or Java, which makes the use of the distributed object model a natural extension of the normal discipline of Object Oriented Programming (OOP).

Distributed Virtual Threads

When working with synchronous method calls in a distributed object model, it is useful to think of a virtual thread of execution visiting different processors as methods are invoked on the objects instantiated on those processors. An important conceptual difference between distributed objects and RPC is that with distributed objects, we stop focussing on where a computation is done. It is done wherever the object is. We will see later why this can be important.

Benefits of a Distributed Object Model

The most obvious advantage of a distributed object model is that it is a natural way to apply object-oriented design to a distributed system, just as RPC is a natural way to apply procedural programming. It is perhaps surprising that a distributed object model offers several other benefits as well. These are ease of management of state, de-coupling of application architecture from hardware architecture, simplified handling of distribution among many processors and simplified failure handling

Remote State Encapsulation

To perform any complex computation remotely, we must store the state of the computation remotely. Objects with methods that modify and use their state are a familiar and natural way of doing this. If we want an RPC-based service to maintain state, we must write RPC functions to allocate memory and return handles, and then have the client pass those handles back to other RPC functions that use them to access memory.

Optimized Local Invocations

The fact that distributed objects (unlike message passing and RPC paradigms) focus on objects as opposed to threads, makes it easy to optimize the case where a computation is performed locally. Instead of sending a message to a local thread, the method can be called directly. If local invocations are optimized in this way, distributed object invocations can be used at a fine level of granularity without incurring a performance penalty.[5]

Decoupling of Application from Hardware Architecture

An important advantage of a distributed object model is that we can defer decisions about where a computation is to be performed until very late in the design. This freedom is very important in the early design stages of a distributed application.

If we work with messages or RPC, we find that we have to define at a fairly early stage in the design where each computation is performed. At this stage, we have an incomplete knowledge of the behavior of the application, so the choice is difficult to make and may have to be revised.

If, on the other hand, we build a distributed object model of the application, most of the design can be done before we decide where to put the objects. When we finally have to locate the objects, the task is easy because the application behavior is well understood.

Distributed Application Portability

This advantage comes into play when the hardware environment changes or the application has to be ported to a different hardware environment. A hazard of distributed application design is that the application can become coupled to the hardware architecture. Because changing the location of an object has little impact on the structure of the distributed object based application, this hazard is reduced. So a distributed object model is a good basis for portability.

Simplified Handling of Computation on Many Processors

Distributed objects make it easier to manage computation distributed among

[5] This optimization is also possible with RPC, but I have not seen it in practice.

many processors. A special strength of this model is the fact that the object references (RORs) are valid on any processor and can be passed from one to another as method parameters.

For example, consider a case where a client on processor A needs computation performed on processor B, using some information from processor X. This would be quite unwieldy with RPC, but with a distributed object model, the client can call a method on an object on processor B, passing a reference to an object on processor X.

Figure 10 shows a complex distribution of computation. Object O1 is on Processor 1 and invokes ROR r1. R1 references O2, also on Processor 1, so it does a local invocation.

This invocation in turn calls a method on r2, which references O3 on Processor 2. The invocation is remote, so it a message is sent using a distributed object kernel.

O3 contains an ROR r3, which references O4 on Processor 3. When r3 is invoked, processing moves to Processor 3, where r4 is invoked and O1 on Processor 1, so processing now visits the object it started from. When each invocation returns, processing moves from Processor 1 back to Processor 3, Processor 2 and Processor 1.

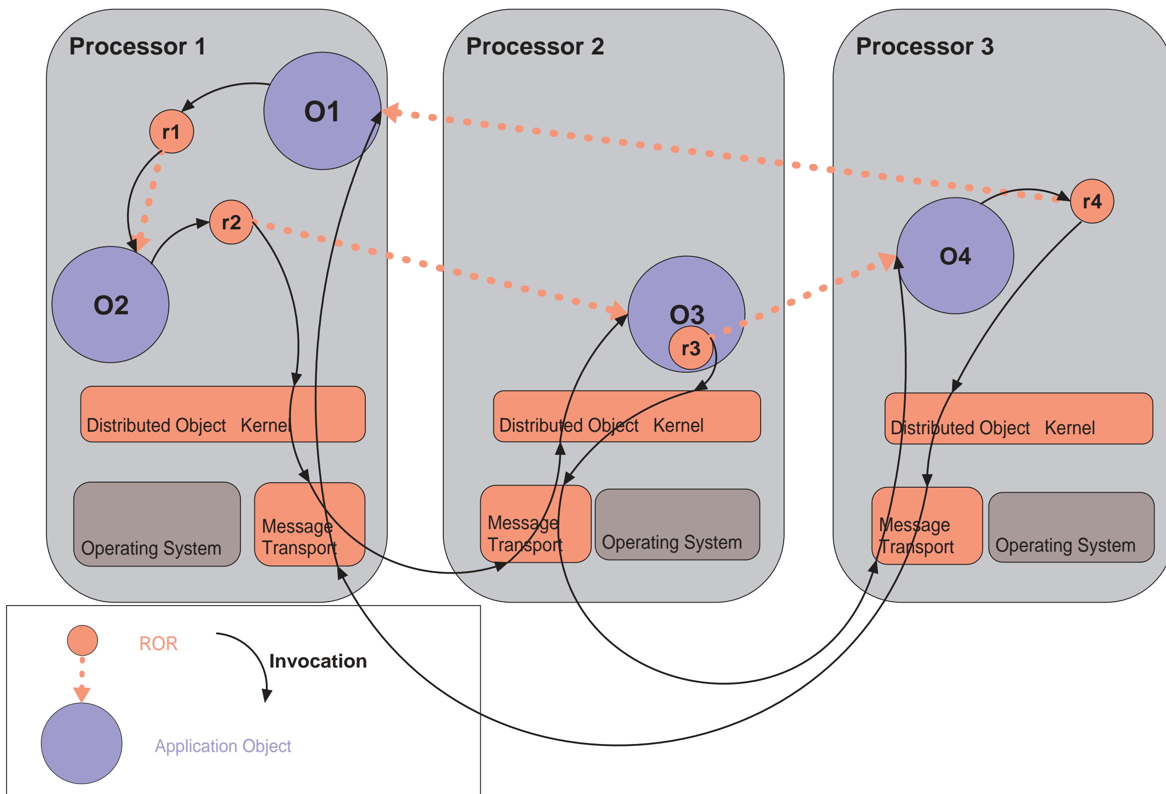

Figure 10: Complex Distribution of Computation

A way to visualize the distributed computation is to imagine a virtual thread of execution visiting different processors as remote invocations are made. It is also easy to see how the location of computation can be controlled by changing where objects are created, leaving the logic of the application largely untouched.

So while RPC makes it possible to link computation on two processors, the distributed object model allows us to easily orchestrate computation on a collection of processors.

Exception Safety - A Free Lunch?

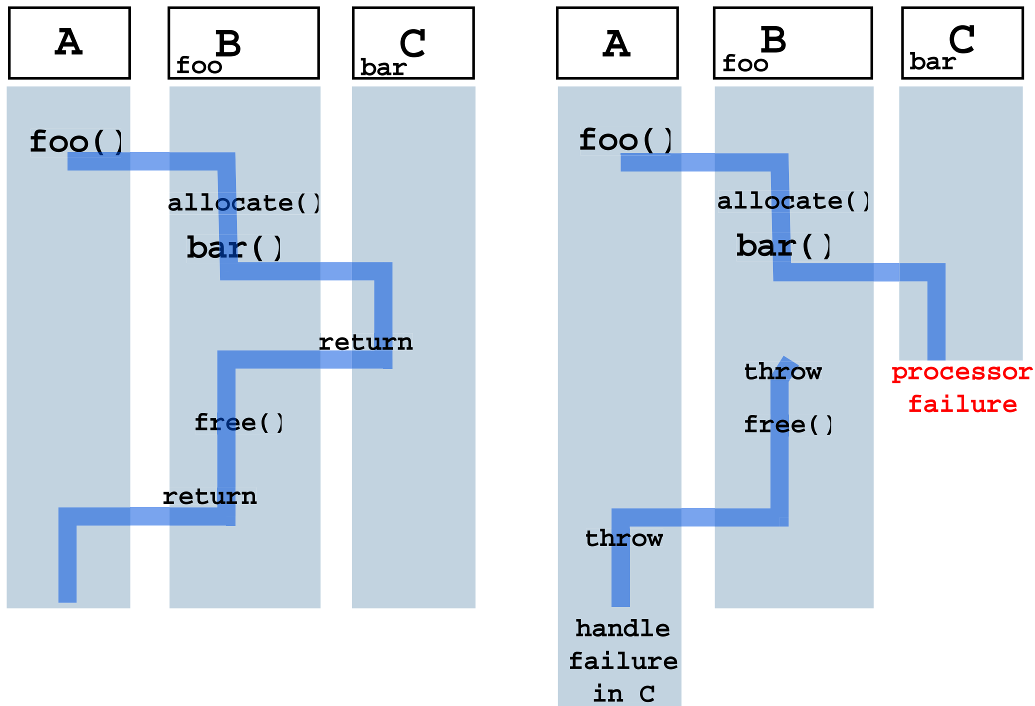

Figure 11: Exception-safe code cleans up after processor failure

Exception handling is a powerful feature of object oriented languages like C++ and Java. Used properly, it allows substantial simplification of design (and coding) so that the essential logic of the application is uncluttered by error handling.

Developers, however, have the responsibility of writing 'exception-safe' code. This means that the code must behave correctly in the presence of an exception. If, for example, memory is allocated in a method, and then freed later in the method, they must ensure the memory is also freed if an exception occurs in between. This is done by catching the exception and freeing the memory in an exception handler (catch clause). The developer practicing exception safe programming does not need to know about the cause or nature of the exception. They just need to make sure that *their* code works right if it happens.

In distributed object models, communication and processor failures are modeled as exceptions. A remote invocation does not return an error code in the event of failure. Instead, it throws an exception. So if we put exception safe code into a distributed application, this code will behave correctly if another processor fails, *even if its developer never considered the possibility of processor failure or even of running in a distributed application.*

Let's take an example. A thread on processor A invokes a method on an object

on processor B, then catches exceptions and deals with any failure. The method on processor B allocates some memory, then calls a method on an object on processor C, then frees the memory and returns control to processor A. If the invocation from B to C fails, this results in an exception on processor B. The method on B (which wasn't specifically designed to handle remote invocation failure) catches the exception, frees the memory and then re-throws the exception. The exception is propagated to the original caller on processor A, which in turn catches the exception and deals with it. The failure has been handled correctly on processor B by code that makes no allowances for distributed systems. This is illustrated in Figure 11. The message sequence chart to the left shows the normal case and the one to the right shows the case where processor C fails.

A drawback of using exception handling to deal with failures is that exceptions have a performance penalty with most compilers. For that reason, some embedded applications are built using a C++ subset that disables exceptions. If exceptions are enabled only to take advantage of failure handling, then this benefit must be weighed against the possible cost of degraded performance. This is why the lunch is almost free.

Distributed Object Technologies

To use distributed object technology in an application, we must choose a specific technology. Options include CORBA[], Microsoft .NET[], Java RMI[], ZeroC's ICE[] and Red Plain Technology's redFOX[]. Of these, CORBA, and particularly Real Time CORBA, has been widely used in embedded systems and redFOX is specifically designed for embedded systems.

CORBA

CORBA is very widely used as a basis for distributed processing in WAN and LAN environments. In CORBA, object interfaces are defined in an IDL specified by the Object Management Group (OMG). An IDL compiler generates C, C++, Java or other language source code for client and server middleware. Each processor has an Object Request Broker (ORB), which does the work of stubs in RPC. A Real Time version of CORBA, called RT CORBA, is used in some embedded systems. Figure 12 shows an OMG IDL definition of a route table.

```
interface route_table {

    typedef uint32_t ip_address;

    bool add(in ip_address prefix, in ip_address mask,
            in ip_address next_hop);

    bool delete(in ip_address prefix, in ip_address mask);
};
```

Figure 12: OMG IDL declaration of route table interface

CORBA typically runs on top of TCP/IP over a LAN or WAN and while it can be adapted to use other transports, it is designed with a high latency transport in mind. With a high latency transport, software overhead is relatively unimportant. If it takes 1ms to deliver a message, it's ok to take 50μs to process it. But if it takes 1μs to deliver a message, then 50μs is a major limitation. CORBA is rich, offering persistence, object migration and other sophisticated features, but these features result in an overhead that is conspicuous in the context of a low latency transport.

With CORBA, a remote invocation can take as long to complete as it takes to execute tens of thousands of instructions. This is large enough to be an important design constraint. Henning and Vinoski [] devote some attention in Section 22.3 to reducing messaging overhead. They say "Naïve IDL design can result in systems that work but are unacceptably slow." and recommend that object interfaces must be modified to reduce the number of invocations needed. Unfortunately, this can distort the application design, reducing the benefits of adopting a distributed object model in the first place.

CORBA's success and shortcomings have inspired alternative distributed object middleware solutions including ICE[] from ZeroC and redFOX[] from Red Plain Technology.

High Performance Distributed Objects

At Red Plain Technology, we are developing redFOX, a high performance distributed object infrastructure specifically designed for embedded systems. It minimizes message size and processing overhead to give performance competitive with hand crafted message passing.

redFOX for C++ realizes the concept of an ROR in a template class dref<T>, which is a distributed reference to an object of class T. These automatically generated classes are small, fixed size, copyable, passable by value, just like C++ references and pointers. This makes them fast to evaluate, manipulate and pass as parameters. Figure 13 shows an automatically generated definition of redfox::dref<route_table>.

```
template<> class redfox::dref<route_table>: public redfox::dref_base {
public:
    typedef uint32_t ip_address;

    bool add(ip_address prefix, ip_address mask, ip_address next_hop);

    bool delete(ip_address prefix, ip_address mask);
};
```

Figure 13: redFOX Remote Interface Specification

THE DISTRIBUTED SYSTEM IN A BOX

Consider a high performance router in a box with a single control processor, eight network card processors and a PCI backplane for inter-processor communications. All processors are 300MHz RISC machines. The backplane is a 66MHz, 64 bit PCI bus. We will contrast this with a more traditional distributed system with a variable number of 2GHz processors on Fast Ethernet (100Mb/s). What aspects are different and how can we exploit these differences? We will address properties that are different in order of importance.

Latency

Bandwidth is the limiting factor for network applications such as web browsing and file transfer, where the volume of data transferred is large. But distributed processing sends small message and waits for a small response.[6] So the main limiting factor on application performance is how long it takes to deliver a single message and get a response, not how many bytes per second can be delivered.

Ethernet has a minimum frame size of 72 bytes. Because frames may be corrupted, the full frame must arrive and have its checksum verified before it can be processed. This means that fast Ethernet cannot deliver a message in less than 5.76µs. If we are working across the Internet, we may have thousands of miles of propagation delay, no matter how much bandwidth we have. Because we cannot transmit messages faster than light, this means we may have a one way latency of 100ms or more.

A PCI bus is quite different. There is no packet, so there is no header. The distance is limited to inches. Transport is reliable, so a receiver can start interpreting a message when its first word arrives. Latency is in the hundreds of nanoseconds and can be under 100ns.

So the “system in a box” provides a much lower latency environment than is usual for distributed systems.

Bandwidth

Fast Ethernet has a bandwidth of 100Mb/s. 1Gb/s Ethernet is now becoming widely available. 66MHz, 64 bit PCI is well established and has a bandwidth of 4Gb/s. PCI itself is approaching obsolescence and there are several contenders for the throne, Hypertransport, Infiniband, RapidIO among them, all with bandwidths in excess of 10Gb/s.

[6] In the case of redFOX, most messages are less than 10 bytes

So our distributed system in a box may have a lot more bandwidth than most distributed systems.

Processor Speed to Latency Ratio

A key metric when evaluating software overhead is the ratio of processor speed to latency. The processor in a box may be 10 times slower than that in a traditional distributed system. This difference magnifies the difference in latency which runs the other way. If we compare 300MHz processors on PCI with 3GHz processors on Fast Ethernet, the former can execute about 100 instructions in the time it takes to deliver a message, while the latter can execute 15,000.

An overhead on message passing that is trivial in a traditional distributed system can be unacceptable in a box.

Reliable Physical Layer

Networks do not typically deliver messages reliably. Packets are routinely lost not only because of transmission errors, but also because they are discarded in a congested network. Reliability must be provided by a transport layer like TCP/IP, which re-transmits packets until they are acknowledged.

PCI and its successors deliver messages reliably as well as quickly. The availability of a reliable physical layer can make distributed processing much easier. It ensures fast, in-order message delivery. Unfortunately, the presumption of unreliability runs deep in distributed systems software, even to the extent that deployed systems run TCP/IP where message loss doesn’t occur.

The contrast is greater when multicasting is considered. Many physical layers including PCI and Ethernet have some sort of multicast capability. But reliable multicast is expensive on an unreliable physical layer. The efficiency of sending one message to many recipients in the physical layer is lost if we have to add a protocol to ensure that each recipient received the message. But if the physical layer guarantees delivery, this protocol overhead is eliminated and multicast can be used to great effect.

Physical Security

Distributed systems are not generally physically secure. It is easy for an attacker to access the network, but not the processors. So there is a need for authentication to prevent an attacker interfering with the system and encryption to prevent information being accessed. Both are a large overhead on message passing.

In a box, however, the communication channel is no more exposed to interference than the rest of the hardware. It is as easy to replace a ROM as it is to interfere with communications. Security can and must be achieved by physically preventing access to the system. So authentication and encryption are unnecessary

overheads.

Number of Processors, Concurrent Startup and Shutdown

In traditional distributed systems, the set of processors in the system tends to vary with time. The application may continue to run while processors come and go. The number of processors involved may be unbounded. By contrast, the number of processors in a box is generally known in advance. Often, they power up and down simultaneously. Protocols for dynamically adding processors to an application, removing them, authenticating them, etc. are not needed.

	LAN OR WAN	**BOX**
Latency	2us-100ms	100ns-1us
Bandwidth	56kb/s-1Gb/s	1Gb/s-100Gb/s
Reliable Physical Layer	No	Usually
Physically Secure	No	Yes
Processor Resources	Large	Small
Number of Processors	Variable, Potentially Billions	Small, Bounded
Concurrent Startup, Shutdown	No	Usually

Table 1: Differences in the Box

Table 1 summarizes the differences between a traditional distributed system and a box discussed above.

CONSEQUENCES

Distributed systems research and products have much to offer the implementer of a distributed system in a box. There is a wealth of knowledge to draw on and there are many wheels that don’t need to be reinvented. But we need to take care to choose the right wheels for our vehicles. Inappropriate use of technology designed for traditional distributed systems will give poor results.

If, however, we consider message latency, processor speed, bandwidth and whether we need to deal with message loss, security and processor unavailability, distributed systems technology and methods will make the transition to working with multiple processors much easier and more productive.

ACKNOWLEDGEMENTS

Thanks to Aidan Kenny, Paul Mannion, Philip O'Carroll, John Roden, Zac Schroff and Sebastian Tyrrell, who read early drafts of this paper. Their comments, suggestions and points of information lead to substantial improvements. The mistakes, however, are all my own work.

REFERENCES AND FURTHER INFORMATION

1. http://www.ietf.org/rfc/rfc1812.txt
2. "Distributed Systems Concepts and Design", Coulouris, Dollimore and Kindberg, Addison Wesley, 2001, Third Edition, ISBN 0-201-61918-0
3. "Distributed Systems", Sape Mullender (Editor), Addison-Wesley, 1989, ISBN 0-201-41660-3
4. "Distributed Operating Systems", Andrew S. Tannenbaum, Prentice Hall, 1995, ISBN 0-13-143934-0
5. www.omg.org/corba
6. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetarch16.asp
7. http://java.sun.com/products/jdk/rmi/
8. www.redplain.com/redfox
9. www.pcisig.com
10. www.ietf.org/rfc/rfc1831.txt
11. http://www.opengroup.org/public/pubs/catalog/c706.htm
12. "Advanced CORBA Programming in C++", Henning and Vinoski, Addison-Wesley, 1999, ISBN 0-201-37927-9.
13. "Binding, Migration and Scalability in CORBA", Henning, CACM Vol. 41, No. 10 (October 1998), pp62-71.
14. http://www.zeroc.com/documents/ieee.pdf

SPEAKER BIOGRAPHY

Dominic Herity is CTO of Red Plain Technology (www.redplain.com). Prior to founding Red Plain, he served as Technology Leader with Silicon & Software Systems (www.s3group.com) and Task Group Chair in the Network Processing Forum (www.npforum.org). He has lectured at Trinity College Dublin (www.tcd.ie), where he contributed to research into Distributed Operating Systems and High Availability. He has many recent publications on various aspects of Embedded Systems design.