APPLYING DISTRIBUTED SYSTEM CONCEPTS TO EMBEDDED MULTIPROCESSOR DESIGNS

PART 1: VARIETIES OF MESSAGE PASSING

By Dominic Herity

In Part 1 of a series published on Embedded.com exploring the use of distributed systems techniques in asymmetric multi-processing, the chief technical officer at Redplain Technology first describes the use of message passing to implement a software feature on a multicore.

After decades of anticipation, multiprocessors are becoming a reality for the majority of software engineers, especially in embedded systems. Symmetric multiprocessors which use a number of identical processors, one operating system and a shared memory are relatively easy to program, because we can use additional processors to execute additional threads.

A different situation arises when we have processors with different instruction sets, different operating systems or without shared memory. In this case, threads running on different processors must use message passing to communicate. This situation is described as “asymmetric multi-processing”. Asymmetric multiprocessors are more difficult to program than symmetric multiprocessors. Software running on an asymmetric multiprocessor must ensure that threads have the data they need when they need it. Results may be affected by the order in which events occur on different processors.

With the advent of asymmetric multiprocessors, the software architect is challenged to design solutions that make effective use of additional processors, but without sacrificing performance and without needing extensive re-engineering when hardware changes.

Asymmetric multiprocessors are effectively distributed systems and some of the tools and techniques of distributed systems can help us address these problems.

This is the first of three articles that will explore the application (and mis-application) of distributed systems techniques to asymmetric multiprocessors in embedded systems. In this, the first part, we will describe an example of a software feature that may need to be implemented on multicore. We will discuss its implementation using message passing and how the semantics of that message passing affect the implementation of the feature.

In the next article, we will discuss the value of higher levels of abstraction built on message passing, specifically remote procedure call (RPC) and distributed objects. We will show how they make software development easier rather like C
and C++ are easier than assembly language programming.

In the final article, we will explore the differences between traditional distributed systems and where we work, in the depths of an embedded system where TCP/IP and Ethernet are as likely to be features as facilities. We will discuss how these differences can mandate technology choices different from traditional distributed systems.

ROUTE TABLE EXAMPLE

I have chosen a forwarding table in an IP router as an example of a software feature to be implemented in an embedded distributed system. I have chosen this example for two reasons. First, the functionality is realistic, but relatively easy to explain and appreciate. Second, it has some of the subtleties that make distributed processing difficult. After explaining this feature, we will discuss its implementation using different distributed processing techniques.

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 directly accessible 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 [1] 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.

```c
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 to table for prefix, mask */

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

Figure 1: Route Table API

The parameter table identifies which route table is referenced. (A router 
may use more than one route table, for example, when implementing Virtual Private 
Networks) 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 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.
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 \( N \times N \) 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 \( N \times N \) 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.\(^1\) The route table protocol runs on the central processor and distributes updates to the replicas.

This is illustrated in Figure 3, below. 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.

\(^1\) The port hardware may use a combination of hardware and software to do packet processing. For simplicity, we will just consider software processing.
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, most packets aren’t 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.
Because the route table has to be replicated at each port, 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 [1] 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 [3] pp 5-7, Mullender says that the possibility of failures in communication and processors is important. This may not be an issue in systems where processors share a bus and a power supply or even a chip, but it sometimes is an issue. Fault tolerant routers are built with redundant hardware that can be replaced while the system is running 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.

Treating the router as a distributed system, we can coordinate route table updates using messages between processors.

VARIATIONS ON MESSAGE PASSING

Message passing seems at first to be a straightforward proposition. A client constructs a message and sends it to a server.\(^2\) The server reads it and takes some action that depends on the content of the message. Part of the action may be to

---

\(^2\) 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 of client and server 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.
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 messages between processors may not be as simple as it appears. Does the sender wait for a response or run ahead? If it runs ahead, how can it collect 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 models 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.

In both cases, 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. The models are different, though, in the semantics of the service they provide.

**Weak Guarantee: Asynchronous, Unreliable, Unordered**

The first model provides weak guarantees and is rather like UDP/IP, but weaker. 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 model does not guarantee delivery of 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 the send function has returned to complete message transmission.

To maintain a route table using the weak messaging model, an API like Figure 4 must be implemented on top of the messaging service. Note that this API is in a layer above the messaging service, not the messaging service API itself. It constructs and sends messages, receives and parses responses and manages message buffers. The messages are received by software on the server that parses
messages, calls functions and constructs and sends responses. This layer of software is called middleware and we'll discuss it in more detail in part 2.

```c
typedef uint32_t ip_address;

int send_route_table_add(struct server* which_server, 
                         void* context, 
                         struct route_table* table, 
                         ip_address prefix, ip_address mask, 
                         ip_address next_hop);
/* Add entry to table for prefix, mask */

int send_route_table_delete(struct server* which_server, 
                            void* context, 
                            struct route_table* table, 
                            ip_address prefix, ip_address mask);
/* Delete entry from table for prefix, mask */

int receive_route_table_response(void** context, void** response);
/* Get message response */
```

Figure 4: Route Table Maintenance with Weak Messaging

This API is much more complex than Figure 1 and it needs some explanation. The functions `send_route_table_add` and `send_route_table_delete` correspond directly to the functions in Figure 1. They construct and send messages using supplied parameters. They have additional parameters, `which_server` and `context`. `which_server` defines the destination of the message - the processor where the route table is located. `context` is an opaque pointer to a user-defined context for the message. This parameter is needed to associate a received response with its corresponding message. It points to whatever information is needed to deal with the response. The function `receive_route_table_response` retrieves a response to a message if any is available and returns a pointer to the response message and the context pointer that was passed with the message. All three functions return an error code if communications fail, but success of the function does not imply successful message delivery.

Clearly, maintaining a route table using asynchronous messages is more complex than calling the C API. Furthermore, the weak message passing model leaves some issues unresolved.

First, 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. But we must distinguish between loss of a message and loss of a response. If we send a message but get no response, we don’t know if the message was processed or not.
Re-sending the message may result in the operation being done twice or more. In the cases of route table addition and deletion, repeating the operation is harmless, but most functions don’t produce correct results if they are called multiple times. Reliable message delivery eliminates this issue.

Second, 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 delivered and 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.

A weak message passing model has hidden costs that are easy to overlook but hard to avoid. These costs are not obvious during software development and testing because failure conditions, processing delays, message loss and out of order delivery generally happen only when a system is stressed. It is necessary to test the application using a deliberately badly behaved version of the messaging service that drops, shuffles and stalls messages at high rates to make sure that the application won’t fail in the field.

**Strong Guarantee: Synchronous, Reliable, Ordered**

Now consider a message passing service with stronger guarantees. A synchronous service sends messages reliably in order. The client side ‘call’ function 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 is processed and a response received. After ‘call’ returns, the client knows that the message has been delivered, processed and answered or that an error has occurred. Figure 5 shows a route table maintenance API using such a strong messaging service. As in Figure 4, we have server parameters and the functions return error codes that indicate communication problems. The context pointer, however, is not needed. Nor is the separate function to receive a response.

typedef uint32_t ip_address;

int remote_route_table_add(struct server* which_server,
    struct route_table* table,
    ip_address prefix, ip_address mask,
    ip_address next_hop, int* result);
    /* Add entry to table for prefix, mask */

int remote_route_table_delete(struct server* which_server,
    struct route_table* table,
    ip_address prefix, ip_address mask,
    int* result);
    /* Delete entry from table for prefix, mask */
Figure 5: Route Table Maintenance with Strong Messaging

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 maintaining context in a specially allocated data structure and without the complication of multiple 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 cost 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 often favored for performance reasons, but is only practical for simple APIs. We need to make sure that like is being compared with like in performance comparisons. If the application is forced to 'strengthen' the messaging service and maintain message context, the result may be more expensive, less flexible and poorer in performance than adopting a well designed strong messaging service to start with.

Because a strong message passing model is easier to use and safer than a weak one, synchronous, ordered, reliable message passing with a response indicating completion of processing will be assumed in the remainder of these articles.

In part 3, we will see that messaging models with strong guarantees provided by hardware are common in embedded systems, but first we will discuss the abstractions we can build on top of message passing and what they do to make distributed processing easier. That is the subject of part 2.

REFERENCES AND FURTHER INFORMATION


SPEAKER BIOGRAPHY

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