integration: a review

ıe class selection and

ican Geographers 71

Geografie 73 323-339

necessities" *Regional*

Statistical Association

-building, partitioning

ment and Planning A

project" *Area* 18 9-13

system for the BBC

h's WEB)" *Computer*

nomic Geography 46

Proc EUUG Conf
Dublin, Autumn, 1987

The HUB: A Lightweight Object Substrate

Michael D. O'Dell

Maxim Technologies, Inc.
Vienna, Virginia USA

1. Introduction

"Lightweight Processes" are currently *très chic* in the leading-edge UNIX community, and "Object Oriented Programming Systems" (OOPS) are gaining in popularity. Both of these ideas may be fundamentally sound, but seem to be suffering somewhat at the hands of hypesters promoting them as the cure for everything from *designus inconceivus* to tertiary code-bloat. Combining these two hot topics is sure-fire way to get funding (and a paper accepted).

The Hub is a simple little operating system which is rather different from most of its process-based brethren seen running about. Like with most systems, these differences characterize both its strengths and weaknesses. The Hub has activities somewhat like processes, called tasks (sorry, but there are only so many words for these things), but it doesn't have the overhead of traditional context switching. Hub tasks communicate primarily with messages, but they don't incur all the message parsing and multiplexing overheads of other systems. The Hub could use a bit of linguistic support which it doesn't get from most programming languages, but the C preprocessor and a modicum of programming discipline largely fill the gap. The computation model of the Hub is somewhat non-traditional to folks raised on the beatitudes of pure processes. And finally, because the basic design doesn't deal with protection domains, the Hub is a *Scout's Honor* programming environment like most little operating systems, and it will run perfectly well within a UNIX process for debugging and development, or as an implementation of user-level "light-weight processes."

It also turns out that the Hub uses some of the central OOPS ideas, admittedly in crude forms, but it is, none the less, an interesting platform for supporting, shall we say, *object-inspired* programming. Hence the title: mix lightweight processes with object-oriented glue, and you get a substrate which can support objecty kinds of things, assuming they don't weigh too much.

2. What's All the Hubbub, Bub?

The Hub was created by Gary Grossman, then with the Center for Advanced Computation at the University of Illinois. The design was first implemented and described by Masamoto [MAS] for his Master's thesis under Grossman's direction. The current implementation is inspired by the system described in Masamoto's thesis, but revised considerably for portability, generality, and more concern for memory management issues.

There are three major elements of the Hub world: the Hub Queue, which gives the system it's name, Hub instructions which are placed in the Hub Queue, and Hub tasks which execute the instructions (see Figure 1). The sequence of operation is really quite

simple: it mirrors the traditional fetch-execute cycle of a computer CPU.

The Hub dispatch loop (the Dispatcher) removes the next instruction from the Hub Queue and inspects it to determine the recipient task and which specific task entry-point should be invoked to execute it. The Dispatcher then makes an indirect subroutine call via a task-specific methods vector supplying the decoded instruction fields as arguments. The task method then runs to completion, adding any necessary new instructions to the Hub Queue. When task method returns, the Dispatcher goes back to the top of the loop to fetch the next instruction waiting in the Hub Queue.

2.1. Instructions Pre-Fetched While You Wait

The Hub Queue functions very much like the instruction prefetch buffer used in modern CPU designs. Instructions which form the next fragment of the instruction sequence but are not yet executing wait their turn in the prefetch buffer, and likewise in the Hub Queue. Where do the instructions come from? In CPUs, the relentlessly advancing program counter marches new instructions into the prefetch buffer. In the Hub system, the stream of new instructions arises from the execution of other instructions! One of the most important side-effects of interpreting a Hub instruction is adding a new instruction or two to the Hub Queue, for without a steady supply of new instructions, *nothing* happens in the Hub world.

Tasks communicate by placing instruction for each other in the Hub Queue. The only way a task ever executes is for an instruction destined for it to bubble to the top of the Hub Queue and be dispatched. For example, instead of the classical *sleep()/wakeup()* process model, an interrupt routine in a device driver notifies its client "top half" of service completion by placing an instruction destined for it in the Hub Queue. With only very minor exceptions, state information is never shared directly between autonomous execution domains (this includes both other tasks and interrupt code). Instead, instructions carrying the necessary information are queued for execution by the recipient. This results in very infrequent processor interrupt lockout, but most importantly, the entire system is manifestly observable, if not truly synchronous. All activity flows through the Hub Queue and the Hub Dispatcher, so one need monitor only that one simple point to produce a very complete picture of exactly what is happening in the system. With a little planning, it is even possible to record and replay instruction sequences to assist in analyzing behavior.

2.2. Detailed Instructions

Hub instructions specify an opcode and several operands: source and destination task identifiers, source and destination port identifiers, and a general-purpose operand which is usually a pointer to a message buffer (see Figure 2). (There can be several general-purpose operands, but rarely is more than one used.) The opcode is equivalent to the method selector of languages like Smalltalk; it identifies which task entry-point (method) is to be executed to interpret the instruction. This saves multiplexing and demultiplexing in several ways. First, we avoid wasting the time spent assembling and then parsing explicit message headers which simply indicate the requested function. Second, decoding the instruction is very straightforward and very fast, and it only need be optimized in one place. Third, task entry-points tend to do only one thing so the path length is minimized and the code simplified.

The source and destination task identifiers in Hub instructions are used to create the *sender* and *self* references used for executing task entry-point methods. A task identifier is a handle to the task's Task State Vector (TSV) as it is called, and tasks are known by their TSV handle. A handle to the new TSV is returned to the parent task when a new task is created.

The source and destination port identifiers are small integers which are available for further multiplexing and demultiplexing within a method. In general, they are simply integers and can take on any such value. Their name, however, arises from an array of port structures in every task's TSV, and port values usually index this array. A port is simply a header for a doubly-linked queue which is included in the TSV port array. These are included in the basic TSV overhead because TSVs often need queues, and having a queue designation encoded in the instruction often prevents reinvention of mechanism and improves observability.

Arbitrary message data can be carried in an instruction by incorporating a reference to it in one of the general-purposed operands. This tends to avoid data copies and provides for quite general messaging. The messages carried by Hub instructions are usually said to flow from the source to the destination port of the participating TSVs.

2.3. Tasks - the Functional Units of the Hub

Tasks are the unit of execution in the Hub system. A task is represented by its activation record, which is called a Task State Vector, or TSV. It is common for "task" and "TSV" to be used somewhat interchangeably, although it isn't strictly correct. Associated with each TSV is a collection of functions called the Task Program (TP) which the task executes in response to instructions. Each such function is called a Task Program Entrypoint, or TPE. Mapping between the instruction opcode and the implementing TPE function is done with an array of function pointers stored in a generic section of the TSV. This binding allows task programs to be shared between tasks even if the code is not strictly reentrant.

The Task State Vector contains two sections: a generic common prefix and a data area specific to the Task Program being executed by the task. The TSV prefix contains the port vector, vectors to the TPEs, and a few other miscellaneous fields. The TP-specific data area is of variable size and is specified when then TSV is created. The *entire* state of the task is represented by data stored in this section of the TSV.

To summarize in object-speak, the Task Program is the methods collection for the object implemented by the task, and since the task program is shared, multiple object instances can be implemented which all share the same methods. Instance variables used by the implementing methods of an object are stored in the object-specific portion of the TSV. Class variables can be implemented (crudely, to say the least) by global variables, or static variables within functions contained in shared TPE code.

The Hub Queue and the Hub Dispatcher, Hub instructions, and Tasks are the basic inhabitants of the Hub world. The way these parts interact to do computations creates a programming environment which is both familiar and strange at the same time.

3. Not Entirely Unlike Processes

This section will examine the programming style which arises from the Hub's unique structure. As can be discerned from the description so far, systems based on the Hub are

composed of a collection of tasks which communicate with each other via instructions, and these tasks implement higher-level abstractions, either abstract data objects or compute objects like protocol machines. In point of fact, the Hub was born to do communications processing. This is a programming area classically described on the blackboard by clouds of processes blithely exchanging messages willy-nilly across beautifully abstract interfaces. The implementation, on the other hand, is usually quite different because of real-world performance requirements. One is easy to understand, the other runs fast enough to be useful. The Hub is an attempt to assuage the disparity between the pictures and the code.

Unmentioned until now is the central notion that all tasks in the Hub world obey essentially the same set of legal Hub Queue instruction opcodes. This makes sense when one reviews the structure of most communications software implementations, particularly those based on the traditional picture with processes. Tasks primarily exchange data with one task on one side, and do essentially the same thing with another task on the other side. The exceptions to this are device drivers, which talk to hardware on one side, but are tasks when viewed from the other, and multiplexing tasks which communicate with potentially many other tasks.

Each Hub Task Program is expected to implement the following basic instructions most appropriately for the function provided by the specific TP.

- **Initialize** - an instruction sent to a newborn task so it may initialize its instance variables before it receives any other instructions. The final act of this TPE is to call a Hub function which acknowledges the initialization. Sending instructions to a task before it has acknowledged the initialization is considered a serious error.
- **Die** - the task should clean up whatever it was doing and commit suicide by calling the appropriate Hub function. Any shutdown synchronization between the requestor and the dying task is purely their business.
- **Timer** - this instruction is posted to a task to notify it that a timer event has expired. The arguments in the Timer instruction are specified when the timer event is scheduled with a Hub primitive function.
- **Data** - this instruction is the basic data transfer mechanism. The argument usually points to a buffer containing the data to be transferred to the destination TSV and port. The usual protocol is that ownership of the buffer is also transferred and becomes the responsibility of the recipient. Note that this is essentially a *write()* function to a task expecting to receive data.
- **Datarequest** - this instruction is used to request a source to send data to the originator. This is essentially a *read()* request from a task desiring data. This instruction may or may not be used in all cases depending upon the flow control protocol between tasks, and the level of asynchrony between the tasks. Packets arriving from a network would typically be posted to an IP protocol machine with a Data instruction, while a task like an FTP server might use a Datarequest instruction when requesting data from its underlying TCP task.
- **Control** - a task-specific control function (like *ioctl()*)
- **Poll** - an instruction for implementing polite busy waiting. Typically, a task waiting for a bit to change in an interface would do the polling by posting a Poll instruction directed to itself. When the Poll TPE gets entered, it checks the appropriate bits and decides whether to post another Poll instruction, or whatever

instruction is appropriate to continue what it was doing. This provides the simplicity of busy waiting without hogging the machine.

- **Debug** - a task-specific function which manipulates debugging state. Hub systems use some internal protocol for determining how the debug state is interpreted, but there is a common mechanism for manipulating it. This encourages useful debugging machinery be included in every TP.
- **Private** - there are two instructions included which are valid instructions, but whose interpretation is local to each TP. This number is easily changed for whatever is needed.
- **Default** - this is not an instruction, per se, but is a TPE which is entered like an instruction whenever an opcode is not one of the above. (The name arises from the C case statement.)

Each Task Program Entrypoint is called from the Hub Dispatcher with two arguments: a pointer to the private area of the appropriate TSV (a *self* context), and a pointer to the instruction which caused this TPE to be executed. This has the unfortunate side effect of requiring local instance variables be addressed with something like

self –> localvariable

and is one area where a little language support would be useful. In Pascal, the *with* clause would do the trick; in C, a few preprocessor **#define**s reduce the pain.

The instruction pointer is supplied so the port values and sending TSVid can be ascertained, as well as to pick up any message buffer pointers in the general-purpose operands.

As can be seen, the Hub is non-blocking. There is no context switching code necessary, and almost no machine-dependent assembler is required, save for that necessary to glue interrupts into the require device driver functions, and for the two functions which set and return the machine's interruptibility state (equivalent to *spl()* and *splx()* in the UNIX kernel). This implies the Hub needs only one stack segment for execution. While a separate stack for interrupt routines an advantageous of some modern processor architectures, there is no requirement for any such support, and it can probably be readily exploited.

4. The Structure of a Hub System

Hub systems tend to be be constructed from "task teams" with one task managing the activity of several worker bees. Manager tasks tend to communicate with each other to create any needed worker tasks and to establish the plumbing between them, leaving the workers to do the actual work. This is much like the *daemon daemon* of 4.3BSD. In an X.25 system currently being built with the Hub, a Level 2 interface manager task watches link devices for signs of life. When the link starts up, the Level 2 interface manager contacts the Level 3 protocol manager with a request for Level 3 service. The Level 3 manager responds by creating a Level 3 protocol machine task and works with the Level 2 manager to arrange the rendezvous between the Level 2 protocol machine and the Level 3 machine. After the initial introductions, the two protocol machines interact with each other and only interact with the managers when some terminating condition

arises.

One important issue in any operating system is flow control. Flow control in this context means the procedures and protocols which constrain the amount of data the system must buffer at any one time. Several recent systems provide for synchronous interactions between processes, thereby rendering the flow control issue essentially moot. However, the disadvantage of this simplification is that the maximum possible concurrency seems to be somewhat limited.

Flow control within a Hub system is visible at two levels. Topmost is the issue of how many Hub instructions are currently awaiting execution. A runaway task which, for example, queues two instructions for each it receives could quickly exhaust the Hub Queue if it proceeded unchecked. This problem is handled by approaching the design with a resource "conservation law" in mind. In other systems, violations of the conservation laws result in bugs like memory leaks, or occasionally freeing an already free resource. The nature of the Hub encourages explicit establishment of such laws as part of the design process (this is both a strength and a weakness).

The other flow control issue is one of how message flow is mediated between tasks. The Hub system provides a modicum of queuing via the instructions in the Hub Queue. Again, a central notion of Hub execution is that a task must always do *something* with a message when it arrives. This is where the port queues come into play. Often, a task can't really do anything with a message because it algorithmically can't proceed; a closed TCP transmit window is a good example. The port queues can be used to sit on the data until the task can dispatch it.

In such a scenario where there is potentially a large impedance mismatch between a stream producer and a stream consumer, an explicit flow control strategy must be used. For example, after a producer sends a buffer in with a Data instruction, it must await a Datarequest instruction from the consumer returning the previously-sent buffer to be refilled. By simply changing the number of allowable outstanding Data instructions to be greater than one, we can easily implement sliding-window style multiple buffering between tasks for bulk throughput applications needing maximal concurrency. In other situations like an Ethernet driver sending packets to an IP protocol module, the interface may not be flow-controlled at all, relying on a simple policy of dropping excess packets upon an overflow condition. The important point is that the level of sophistication needed by any particular interface can be crafted from the available Hub facilities, thereby neither overbuilding for some uses or underbuilding for others.

In concluding the discussion of the Hub environment as seen from inside, it should be said that the Hub was intended to be a flexible framework for implementing what is needed to do the specific job at hand. It has a modest set of facilities beyond those described like timer management and buffer and storage allocation which, when taken with the Hub facilities described above, form more of an operating systems toolkit, rather than an ornate edifice replete with strictures. This is, of course, a double-edged sword.

5. Closing Notes

There are two other areas which deserve comment, one because it was advertised, and another because it is interesting to consider in light of the current architectural trends.

5.1. Some Thoughts on Implementing Objects

I hope by this point the notion of using the Hub to implement objecty systems, possibly hidden behind some syntactic overcoat, is not completely absurd. There are more than a few problems if you want real Smalltalk, but with a modicum of reserve, the Hub can go a reasonable job of supporting object-style programming done in a more traditional programming language. Note that this opinion arises from the belief that the most important feature of object-oriented programming is one of encapsulation, with limited, static inheritance far short of the Smalltalk extreme probably being quite adequate to realize the advantages of OOPS for most tasks. Others will certainly differ, probably strongly. That's what makes horse races.

The other topic has to do with the notion of context switching in general and the impact of evolution in machine architectures on the basic complexity of this mechanism which is so central to traditional process-rich systems.

5.2. Context Switching and RISC Machines

The traditional process context switch involves saving the "visible" processor state (register values, condition flags, interrupt level, floating point modes, memory management state, etc.), and reloading the processor with a new copy of this same information. In its full glory, a great many bits move around, and many, many machine cycles can go by if the state is very complex.

RISC architectures are often characterized by large register stacks on the processor chip which must be loaded and unloaded on context switches. These stacks are generally much larger than those of their CISC friends and it is interesting to ponder the performance impact of heavy context switching upon RISC performance. One useful description is that a process-based system tends to be "broad," meaning it spans many different state domains (many relatively shallow stacks), while the state of a RISC machine tends to be "deep," meaning it excels at nested subroutine calls within one state domain (stack). Since the Hub systems needs only one stack, and spends all its time doing subroutine calls, it may be particularly suited to RISC architectures. This is an interesting area to pursue experimentally with actual measurements!

6. You Have Tea and No Tea

In conclusion, the Hub is both traditional and radical at the same time. It is both process-like and thread-like. It contains some objecty notions, and some explicit programmer responsibility for managing the environment. Maybe the way to see the landscape is as a continuum between process-richness at one end, and traditional monolithic realtime multi-threaded systems at the other, with the Hub as an operating point somewhere in between. Traditionally, a *process* is a defined, fabricated object which, like Algol, everyone understands but can't quite nail down. With the advent of Objects, which seem to be somewhat like processes in their persistence and activity, but which somehow don't really satisfy the intuitive definition of *process*, maybe *process-ness* has become an analog value which can be possessed in greater or lesser degrees instead of being an absolute attribute. This is certainly in the spirit of the Hub, and possibly places the Hub at the confluence of Object-oriented programming concepts and operating systems.

7. Bibliography

[MAS] Masamoto, K., "Implementation of HUB Processor," Master's Thesis, University of Illinois at Champagne-Urbana, 1976

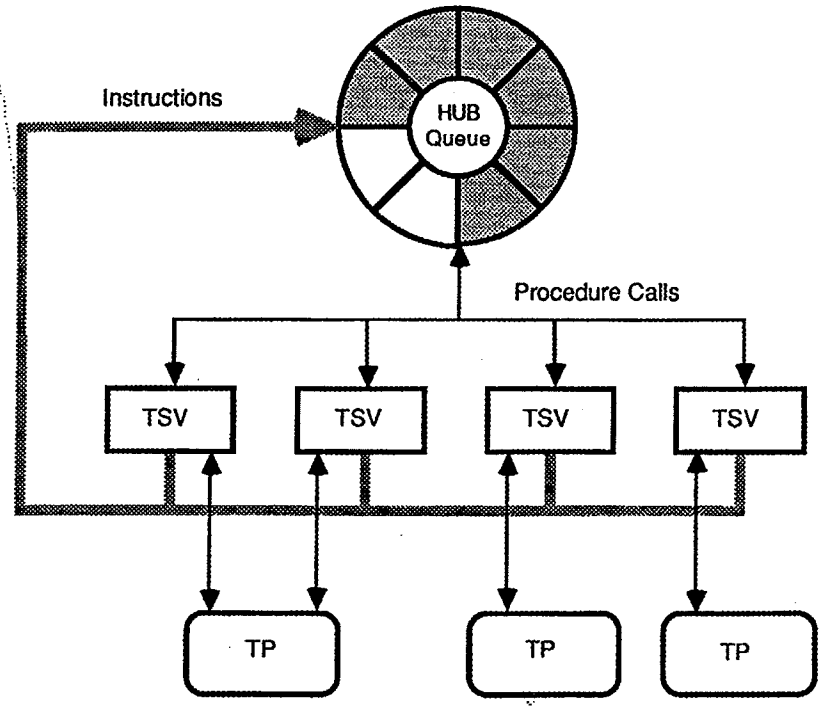

Figure 1
The HUB System

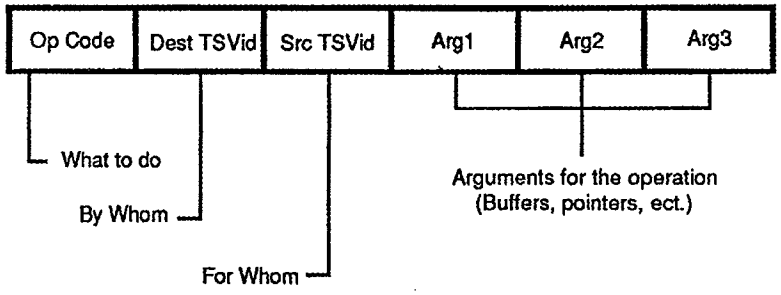

Figure 2
A HUB Instruction