Using Dispatchers to Control Process Structures

Darrell R. Raymond

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1

ABSTRACT

Multiple process structures are an important technique for modularizing applications, but good interprocess communication requires careful design. This paper describes a new process model called the *dispatcher*, which simplifies communication by mediating the transfer of messages between processes. Dispatchers manage courier processes and hence support a type of non-blocking communication which is useful in systems that provide only blocking primitives. Other advantages of the dispatcher are its facilitation of incremental development and support for well-defined process interfaces. Experiences with the dispatcher are described, and some extensions are presented.

June 11, 1987

This research was funded by grant G1154 of the Natural Sciences and Engineering Research Council of Canada.

Using Dispatchers to Control Process Structures

Darrell R. Raymond

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1

1. Introduction.

Modern operating systems encourage the use of multiple intercommunicating processes for applications programming.[1-3] These systems support efficient process creation, scheduling, and communication, thereby increasing parallelism and easing the transition to distributed environments. An important advantage is that multiple processes are often a powerful means by which to decompose and modularize applications.[4,5] However, these advantages are offset by the need to solve new development problems such as scheduling, deadlock, prioritization, and allocation of concurrency. Proper resolution of these issues entails a significant amount of design effort.

One recommended technique for design of multiple process structures is to employ anthropomorphic models;[6] some examples are the *administrator*, *courier*, *worker*, and *vulture*.[7,8] This paper describes a new model that simplifies interprocess communication — the *dispatcher*. Dispatchers mediate a form of non-blocking communication, and so are useful in systems which only support blocking communications primitives. The dispatcher evolved partly in response to perceived complexity in existing models, and partly as a more robust (though less efficient) type of process of particular utility in developing user interface applications. Experience with the dispatcher shows that it can lead to well-designed process modules, and so has advantages in addition to managing non-blocking communication.

Before describing the dispatcher in more detail, we first briefly describe the two main types of message passing primitives.

2. Blocking and non-blocking primitives.

Many applications can be modularized into independently executing processes that communicate by means of messages. Generally, these messages are one of two types: those which block the invoking process, and those which do not. Blocking primitives enforce a *rendezvous* of two processes; each process stops at a specified point in its execution, and then data is exchanged. For example, suppose that two processes `x` and `y` wish to exchange data. Process `x` can request a rendezvous with process `y` by invoking `send(message, return_message, y)`. At some point in its execution `y` permits the rendezvous by invoking `receive(message, x)`; this may have occurred before `x` invoked the `send`. When rendezvous is achieved, `y` is unblocked and the message `message` is transferred; `x` remains blocked until `y` invokes `reply(return_message, x)`. `x` then obtains `return_message` and can continue processing. Non-blocking primitives do not require rendezvous of processes; a non-blocking `send` results in kernel buffering of the message until the recipient attempts to read it. Similarly, a non-blocking `receive` simply checks to see if a message has been buffered. The tradeoffs between blocking and non-blocking primitives and the `send`— `receive` — `reply` cycle have been described at some length by Gentleman.[8]

Each type of primitive is natural for specific classes of communication. Blocking primitives using a `reply` message are natural for situations where data must be transferred in both directions; for example, a request for disk data should be answered with the data, and the requesting process should be blocked until the data is received. Non-blocking primitives are natural for situations where communication is one-way and a return value is unnecessary. For example, a process generating keyboard input events need not wait for a return value. An additional distinction is the responsibility for message buffering. Non-blocking messages are buffered by the kernel; blocking messages are buffered by the application.

Several operating systems provide only blocking primitives; examples include PORT,[9] Harmony,[10] and MINIX.[11] The robustness and simplicity of blocking primitives are of overriding importance in the operating system kernel, where buffering is especially costly and the unbounded queue problem especially troublesome. However, the natural applications of two-way and one-way communication suggest that applications programmers should be able to employ either type of communication. As a result, programmers often simulate non-blocking communication within a blocking-only system using couriers.

3. Couriers.

Couriers are used to transmit messages when the originator of the message does not wish to be blocked until the intended recipient is ready. The code for a typical courier is seen in Figure 1, written in a C-like language. The courier repeatedly **sends** to its creator for a packet, extracts the message and the destination from the packet, and **sends** the message to the destination. When the destination process is ready, it receives the message from the courier, and may give it a message to be returned to the sender. When the courier has received the message and destination (but before the recipient has received the message), the sender is unblocked and hence free to continue processing. The courier now blocks on the recipient until it is able to transfer the message. In effect, a courier provides a high-level operation similar to a non-blocking **send** by "buffering" messages within processes.

```
#include <packet.h>

courier()

{
  int        ready;
  struct     packet   who_and_what;
  Pid        who;
  char       *what;

  repeat
    {
     send(ready, who_and_what, Creator's_id);
     who  = who_and_what.who;
     what = who_and_what.what;
     send(what, ready, who);
    }
}
```

Figure 1. A courier.

Though the implementation of the courier itself is simple, the impact of couriers on an application can be complex. For example, the number of couriers in the system is important. If too few couriers exist, then some processes will be delayed while needed couriers are created. If too many couriers exist, the performance of the application or even the operating system may be degraded. Furthermore, processes which employ couriers suffer extra overhead in courier creation and destruction, in maintaining a queue of ready couriers, and in distinguishing between messages from other processes and the returning couriers. All these factors tend to discourage the use of couriers, and hence of non-blocking communication.

4. Dispatchers.

The need for courier management led us to the notion of the *dispatcher*. A dispatcher is a centralized courier service for a given set of processes. A process wishing to transfer a message obtains a courier from the dispatcher; the courier delivers the message and then reports back to the dispatcher for further messages. The dispatcher creates its couriers, manipulates the queue, and handles the return status of the couriers, thus freeing the application processes of courier overhead. Since courier management activities are quite general, it is both reasonable and useful to handle them in an application-independent fashion.

```
#include <requests.h>
#include <msg.h>

request( activity, data, destination )

requests activity;
char     *data;
Pid      destination;
{
  struct     message msg;
  int        junk;

  msg.pointer = data;
  msg.req_type = activity;
  msg.pid = destination;
  send(msg, junk, dispatcher_id);
}
```

Figure 2. `request`.

In a dispatcher-oriented environment, an application consists of a set of processes, a set of couriers, and a dispatcher which manages the couriers. Any process can be the source or sink of a message, but messages are not sent or received by direct use of `send` and `receive`. Instead, a process that wishes to send a message invokes a `request`; a process wishing to receive a message invokes a `get_request`. These calls entail the use of a courier for data transfer.

The form of the `request` call is shown in Figure 2; a `request` instructs process `destination` to perform `activity` with parameters `data`. A `request` is non-blocking, so the invoking process is permitted to continue with its execution after the `request` has been noted. However, the invoking process may not assume that the recipient has immediately serviced the request. All that is guaranteed is that the request will be eventually serviced, and that successive `requests` from the invoker will be serviced in the order in which they are made. `requests` from several processes to one are not serviced in a specific order unless order is imposed by the recipient.

Within the **request** function the arguments are assigned to the appropriate components of a record, and then sent to the dispatcher. Code for the dispatcher is seen in Figure 3. Initially the dispatcher creates a set of couriers and appends them to its ready queue; then the dispatcher blocks while attempting to receive a message. If the message is the result of a **request**, the dispatcher allocates a courier from the ready queue, transfers the information to the courier, releases the courier, and then **replys** to the requestor, hence unblocking it. If the message is a notification of a returning courier, the courier is appended to the ready queue.

```
#include <courier.h>
#include <msg.h>

dispatcher()

{
  couriers    *courier_queue;
  struct      message      msg;
  int         junk;
  Pid         requestor;
  Pid         courier;

  create_couriers();
  repeat
    {
     requestor = receive_any(msg);
     if (a_courier(requestor))
       append_to_queue(requestor);
     else
       {
        courier = head_of_queue();
        assign_courier(requestor, courier, msg);
        reply(junk, requestor);
       }
    }
}
```

Figure 3. Dispatcher.

If there are no couriers available when a **request** is made, the dispatcher has several options. In the example implementation we assume that **head_of_queue** will create a courier if necessary. Other possibilities are to note the **request** and let the requestor proceed, or to leave the requestor blocked until a courier is available. This latter possibility might be termed *soft blocking*: a **request** is not blocked unless no resources are available for transmitting the message.

A process that wishes to receive messages invokes get_request. The code for get_request is seen in Figure 4. get_request blocks the invoking process, so it is usually employed only when the process has completed servicing all other computation. get_request waits for a courier to attempt rendezvous, then the message is extracted and copied into a buffer (whose address was passed by the invoking process) and the courier is released to return to the dispatcher. The return value of get_request is the desired activity.

```
#include <requests.h>
#include <msg.h>

get_request( data )

**char   data;
{
  Pid         pid;
  struct      message msg;
  request     activity;
  int         junk;

  pid = receive_any(msg);
  activity = msg.request;
  data = msg.data;
  reply(junk, pid);
  return(activity);
}
```

Figure 4. get_request.

Processes which use the dispatcher's services typically assume a server-like form as shown in Figure 5. Each process consists of a main routine which repeatedly invokes get_request and directs these requests to the appropriate function. In servicing requests a function may (and typically does) issue new requests which are serviced by some other process. get_request is invoked only as shown in the main module of the process, and not in its other functions. This last constraint ensures that the flow of messages into a process can be examined simply by reading the main routine.

The main benefits of dispatcher-oriented process structuring derive from the indirect use of message-passing primitives and flexible development of process structures. The basic disadvantage of dispatcher-oriented process structuring is the efficiency penalty of transmitting messages with couriers.

Parallelism is more easily achieved in a dispatcher-oriented environment due to its indirect use of message-passing primitives. Careless use of blocking primitives can result in an overconstrained order of execution (i.e., subroutining) and hence very little parallelism. However, the use of a **request** blocks a process only for the length of time necessary to obtain a courier, so subroutining is avoided. Furthermore, communication is more robust if the primitives are used indirectly. If message-passing primitives are used directly and a message of incorrect type is received, the sending process is destroyed. Similarly, **sends** to non-existent processes cause the death of the sender. In the dispatcher-oriented environment, a courier is destroyed rather than the requestor. The use of couriers for communication implies an acyclic blocking graph, hence simple deadlock is avoided. Because of the non-blocking nature of the **request**, two processes can **request** activity of each other simultaneously without simple deadlock, and in fact a process can make a **request** to itself.

```
#include <requests.h>

main()

{
  char        *data;
  requests    activity;

  initialise();
  repeat
    {
      activity = get_request(&data);
      switch(activity)
        {
         case type_1  : service_type_1(data);
         case type_2  : service_type_2(data);
        .
        .
        .
         case type_n  : service_type_n(data);
        }
    }
}
```

Figure 5. Main function of a process.

One other comment about standard message-passing primitives is that their names are a potential source of misunderstanding. A process invoking a `send` would seem to be the active initiator of a dialogue, and the invoker of a `receive` seems to be the passive acceptor of dialogues. Programmers might also reasonably conclude that `send` is used when information is to be transmitted, while `receive` is to be used when information is obtained. These perceptions are understandable because of the normal connotations of the words "send" and "receive". However, an administrator[8] initiates dialogues and transmits information without employing a `send` (instead, it `replys` to a waiting courier with the data to be sent). `send` and `receive` can be considered identical in that they indicate the desire for rendezvous; their most important difference is that the process invoking `receive` is given control during the rendezvous. The primitives `get_request` and `request` have the psychological advantage that their connotations closely resemble their semantics.

The non-blocking nature of the `request` facilitates incremental development of process structures. Blocking primitives interfere with incremental development because processes may communicate only in certain restricted ways. This constraint is removed in the dispatcher-oriented environment, so the programmer may freely add or delete processes and `requests` without fear of deadlock. Furthermore, the robust nature of the messages suggests that an application can be tested before all its processes have been developed. Processes are not blocked if their `requests` are not served, nor do they need to service any `requests` they receive, so an incomplete application will not suffer process structuring problems. For example, an application which has processes for input, screen display, and file output can be created with only a "stub" file output manager. `requests` to the file output process need not result in any file output, but the stub permits the remainder of the process structure to be implemented and tested. Functionality can also be added to the stub incrementally by coding the activity to be taken for each `request` separately.

5. The name service.

A preliminary implementation of the dispatcher was developed in PORT V2.3 on an IBM PC/XT in order to test the viability of the model. The major problems we expected were significant performance penalties and possibly a variant of the unbounded queue problem i.e., exhausting the list of couriers. While the use of couriers certainly increased the cost of message-passing (and hence overall system load), we were agreeably surprised to find that dispatcher-mediated communication was effective except for tasks which would generate many messages that must be handled in real-time, such as tracking a mouse.

Processes must specify the destination of their `requests`; as shown in Figure 2, this implies that the process id of the destination is known. We found this need to know the process ids of message recipients troublesome for two reasons. First, process ids are dynamic and hence must be obtained at run-time, necessitating extra programming overhead. Even if it were simple to obtain this information, it seemed that such low-level knowledge reduced the modularity and robustness of the application. Consequently we decided to enhance the dispatcher by adding a *name service* that would map symbolic process identifiers to absolute process ids. Given such a service, the processes no longer need to know absolute process ids; `requests` can be made with the symbolic process identifiers, which are resolved by the dispatcher at run-time.

In this enhanced dispatcher environment an application is started by creating its dispatcher, whose first activity is to create the couriers and processes involved in the application. The processes to be created are listed in a *startup file*, which contains the location of the object file of each process, and a symbolic name by which the other processes can address it. The names and the associated process ids are entered into the name service, with the dispatcher ensuring that there are no name collisions. Next, the couriers are created; in the current implementation of the dispatcher one courier is created for every process specified in the startup file. Another possibility is for couriers to be created as they are needed.

Once the application's processes have been created and the name service has been initialized, the dispatcher mediates communication between the processes as before. The implementation of `request` is changed slightly since the dispatcher can now be referenced as `Creator's_id`; hence even the dispatcher's process id can be known indirectly. Destination processes are referred to by their name rather than their (unknown) process id. When the application is finished one of the processes sends a termination `request` directly to the dispatcher so that the processes and couriers can be destroyed in an orderly way.

The use of symbolic names for processes enhances the dispatcher's ability to support incremental development. Processes can be added or deleted simply by modifying the startup file, without the need to recompile or otherwise change the other processes. This facility is commonly used in testing and comparison of alternative processes. For example, a database query language process might include a request for `search` to a process `datastructure`. If the programmer has coded data structure processes `Hash`, `B-tree`, and `SortedList` so that they each handle `search` in the appropriate manner, they can each be tested simply by editing the startup file so that the desired process has the appropriate symbolic name.

When processes are built using the message-passing primitives and process ids directly, there is too much opportunity for the creation of non-modular, application-dependent code. By contrast, the dispatcher and name service environment foster a modular form of process development. In this environment each process presents a highly abstract face to the rest of the system, consisting only of its symbolic name and the set of activities it services. The dispatcher explicitly enforces and supports the development of modular process structures, promoting software reusability and information hiding. We have found these characteristics especially useful for rapid prototyping of user interfaces and software for psychological experiments.

A production dispatcher with name service has been implemented in PORT and used for the development of several applications by the author and research assistants. A `request`— `get_request` cycle averages 25.2 milliseconds, in comparison with a `send`— `receive` — `reply` cycle under similar conditions which requires 6.2 milliseconds. The dispatcher is more than four times slower than the direct messaging method.

The first application was a graphical interface to hierarchical file system. The interface was composed of five processes which handled input, process state, data logging, graphical display, and the data structure. The interface has undergone two revisions, and has served as the basis for software used in three different user interface experiments. The use of the dispatcher in each of these applications resulted in a significant decrease in overall implementation effort. A second major implementation was a query mechanism for a videotex database that employed Venn diagrams for specification of boolean keyword expressions.[12] This application was developed concurrently with the graphical interface and employed the same input and graphical display processes with a small extension in functionality. Additional processes were developed for keyword list management, display of query results, and management of venn diagrams. Currently the file system interface and the keyword interface are being merged.

6. Extensions.

The dispatcher has proved to be a worthwhile software utility for developing and maintaining multiple process applications. During implementation we noted and explored some opportunities for extended functionality.

The first extension resulted from the observation that the centralization of messages in the dispatcher can be exploited for debugging purposes. Since all of the application's `requests` pass through the dispatcher, it is a vantage point for observing the execution of the process structure. Accordingly, we modified our implementation to include a simple "debugging" mode, in which each `request`'s source, destination, and request type are displayed in a debugging window. By following the messages in the debugging window the programmer can trace the application's activity and observe erroneous or ignored `requests`, or detect runaway processes. The programmer can also interrupt the dispatcher at any point, thus halting the application temporarily. These basic facilities could be augmented in several ways, including the provision of a "single-message" capability (similar to single-step in a standard debugger), dynamic modification of process priorities, dynamic addition or deletion of couriers, and the ability to insert specific `requests` into the application by hand.

Another interesting extension results from the observation that the modular interface of dispatcher processes promotes software reusability. Processes in a dispatcher-oriented environment are highly independent entities with a well-defined interface. As a result, programmers tend to build processes which have general utility. For example, we built a `Log` process which maintains a file of logged information. `Log` accepts requests to open a named file, append a line of data, and close the file. Any application which wishes to maintain a log can include this process in its startup file, along with the appropriate identifier (i.e., the one to which log `requests` are made by the application's processes). Furthermore, any number of log files can be handled by simply adding extra instances of this process in the startup file. `Log` and other such processes can be added to a general process library, thus significantly reducing the time to develop new applications. Programming in a dispatcher-oriented environment bears some resemblance to Unix shell programming, in that both the shell and the dispatcher are used to connect general processes to perform some specific task. The dispatcher is more primitive than the Unix shell in that it is not programmable, but it is more powerful in that its "pipelines" can be non-sequential. This is an appealing view of process structuring, but proper exploitation of the idea requires a more capable tool than the dispatcher.

Several other extensions have been briefly investigated. A dispatcher could be designed to handle `create` and `destroy` requests so that the application's components could be added or deleted as necessary. This might be important if an application was quite large or only required some processes at specific points in its lifetime. Another extension would be to define a `rename` request that changes the names of one or more of the application's processes (and hence the message flow). Lastly, since a dispatcher is indifferent to the activities of the processes it mediates, it can manage other dispatchers. Since the current dispatcher generates no `requests`, this would merely be a way to control the creation of a multi-dispatcher process structure. A more powerful facility would result if dispatchers could pass messages to parent dispatchers. Such a system might be used to implement an inheritance hierarchy similar to that in object-oriented systems, with nested dispatchers providing some services and passing unknown requests up to their parent dispatchers. We have not yet implemented a hierarchy of dispatchers.

7. Conclusions.

The dispatcher was originally intended to provide non-blocking communication within a system that supports only blocking primitives. However, the abstraction provided by the name service would be a useful facility even in a system which supports non-blocking primitives. Hence the dispatcher contains the core of a more general process structuring tool.

Experience with dispatchers suggests that it is unsatisfactory to merely teach process structuring models without providing explicit programming support. Existing process models are typically presented as ideas, but the dispatcher is most valuable as an implemented utility that significantly reduces the amount of coding required to develop multiple process structures. Modern programming languages encourage good control flow structures by making proper structuring easy; similarly, multi-process structuring systems should encourage good process structures by making proper process structuring easy. This type of environment is especially suitable when the primary motivation is teaching or the application requires incremental development.

8. Acknowledgements.

I would like to thank Frank Wm. Tompa for permitting me to undertake this work, and for providing the PORT workstations used in the implementations. Vilhelm Böggild participated in many discussions about process structuring and developed the keyword system largely on his own initiative. Bert Bonkowski of Waterloo Microsystems Inc. provided insight into the issues, along with helpful advice about programming in PORT. Thanks also to Jay Black for his advice and comments.

9. References

1. Cheriton, David R., Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager, Thoth, A Portable Real-Time Operating System (revised), CS-78-15, Department of Computer Science, University of Waterloo, Waterloo, Ontario (March 1978).
2. Cheriton, David R. and Willy Zwaenepoel, One-to-Many Interprocess Communication in the V-System, STAN-CS-84-1011, Department of Computer Science, Stanford University, Stanford, California (August 1984).
3. Forsey, D.R., Harmony in Transposition: A Toccata for VAX and Motorola 68000, M.Math Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario (March 1985).
4. Beach, R.J., J.C. Beatty, K.S. Booth, E.L. Fiume, and D.A. Plebon, The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program, *Computer Graphics* **16**(3) pp. 277-287 (July 1982).
5. Tanner, Peter P., Marceli Wein, W. Morven Gentleman, Stephen A. MacKay, and Darlene A. Stewart, *The User Interface of Adagio, a Robotics Multitasking Multiprocessor Workstation,* IEEE International Conference on Computer Workstations, San Jose, California (November 1985).
6. Booth, Kellogg S., Jonathan Schaeffer, and W. Morven Gentleman, Anthropomorphic Programming, CS-82-47, Department of Computer Science, University of Waterloo, Waterloo, Ontario (February 1984).
7. Cheriton, David R., Multi-Process Structuring and the Thoth Operating System, CS-79-19, Department of Computer Science, University of Waterloo, Waterloo, Ontario (May 1979).
8. Gentleman, W. Morven, Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept, *Software — Practice and Experience* **11** pp. 435-466 John Wiley & Sons, Ltd., (1981).
9. Port Language Documentation, DEV 1.10, Waterloo Microsystems Inc., Waterloo, Ontario (January 17, 1986).
10. Gentleman, W.M., Using the Harmony Operating System, ERB-966 (NRCC No. 24685), National Research Council of Canada, Ottawa, Ontario (May 1985).
11. Tanenbaum, Andrew S., *Operating Systems: Design and Implementation,* Prentice-Hall, Englewood Cliffs, N.J. (1987).
12. Böggild, Vilhelm, Graphical Specification of Keyword Queries to Videotex Databases, CS-86-63, Department of Computer Science, University of Waterloo, Waterloo, Ontario (November 1986).