SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 1, 59–79 (1989)

Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime

M. E. SEGAL
EECS Department
University of Michigan
Ann Arbor, MI 48109, U.S.A.

O. FRIEDER
Bell Communications Research
445 South Street
Morristown, NJ 07960, U.S.A.

SUMMARY

When a new version of a software system is created, the current version of the system must be shut down while the new version is installed, resulting in software 'downtime'. There are application domains where such downtime is undesirable if not wholly unacceptable. A dynamic program updating system replaces, or updates, a computer program with a new version while the program continues to run. A dynamic program updating system for programs written in conventional procedural languages such as Pascal and C is described. The proposed system updates programs without causing substantial performance degradation and requires minimal user intervention to initiate the update. A fully functional prototype updating system is presented and a sample program, namely an internet packet router called the 'Packet Pumper', is updated. The performance of the updating system and the Packet Pumper is discussed.

KEY WORDS Dynamic program updating Dynamic code replacement Distributed systems Software downtime

Software maintenance research has traditionally addressed the question 'What can researchers, software engineers, programmers and managers do to make programs more adaptable to the changing environments in which they must exist?' Adaptability (and thus maintainability) encompasses many issues including good initial program design, well-written program code, powerful development and maintenance tools, adequate documentation, and supportive management to encourage proper use of these resources. One question that has not been given substantial attention in the software maintenance literature is 'Having used the available software maintenance tools to enhance a program,

[1] 'Environment' encompasses both the tasks that the program must perform and the interactions with other (possibly real-world) entities.

1040–550X/89/010059–21$10.50
© 1989 by John Wiley & Sons, Ltd.

Received 16 January 1989
Revised 14 March 1989

how can a program be introduced into an environment[1] without disturbing that environment?' In many cases, this can be accomplished by merely terminating the old version of the program and starting the new version. Unfortunately, there are programs where halting the old version and loading the new version is unacceptable, primarily due to the high cost of such an operation. This cost can manifest itself in lost revenue (for example, airline reservation systems and telephone switching systems) or in danger to human life (for example, nuclear reactor control, computerized life support and air traffic control). The ability to *dynamically update* a program, i.e. load a new version of a program without stopping the currently running version, could alleviate these costs in many cases.

A scalable, distributed, dynamic program updating system is presented and evaluated. The described system updates a running computer program written in a standard procedural programming language such as Pascal or C, without substantial degradation in the performance of the program being updated. The updating system presented provides mechanisms for ensuring consistency during the update. The system executes on most modern computer architectures and does not require special-purpose hardware. Assumed is that programs are written using a top-down design methodology with appropriate encapsulation of abstract objects contained in the program.

The remainder of this paper is organized as follows. In Section 1, a brief summary of previous work in this area is provided. Section 2 reviews the various aspects of the updating system initially described in Segal (1988). The types of programs that can be updated using the proposed system, and the criteria used for performing the update are described. Section 3 discusses an example scenario from the computer networking domain where the use of a dynamic updating system is beneficial: changing the routeing algorithms of an internet packet router. Section 4 overviews a prototype updating system that was constructed to test the ideas presented in this paper and demonstrates how it would be used to update a hypothetical packet router called the 'Packet Pumper'. Packet Pumper performance during the update is discussed in Section 5, and concluding remarks are presented in Section 6.

1. RELATED WORK

The problem of replacing portions of computer programs without stopping them has been examined in the literature by several different researchers. The previous work can be classified into three main categories, as discussed below. *Hardware-based approaches* attack the problem by providing a redundant CPU and peripherals to be configured with a new version of the program while the old one continues to run (Rey, 1986; Schell, 1971). When the program is updated, the old system is physically disabled while the new one is enabled. This approach has the advantage of providing hardware fault-tolerance, but it does so at a substantial cost.

Software-based service-oriented approaches attack the problem by imposing a server/client relationship on the programs they can update (Bloom, 1983; Bodwin, 1987). In such an approach, a number of *clients* request a service from a *server* via some well-defined mechanism such as an operating system primitive or a remote procedure call (Birrell, 1984) if the clients and servers are distributed as in Bloom (1983). A server may be updated by temporarily disabling its services and then installing a new server. Although this approach will work in a distributed system, it will only work with software systems that observe a server/client relationship.

Finally, *software-based procedure-oriented approaches* attack the problem by replacing individual procedures as the program executes. In such an approach, when all of the 'old' procedures have been replaced by all of the 'new' procedures, the program has been updated. This class of updating system is related in some respects to *dynamic type replacement systems* such as Fabry (1976). In a type replacement system, the routines providing access to abstract data types are replaced while the program using them continues to run. Although this type of system allows abstract data type implementation to be changed between versions of a program, it does not address the more general issues of code restructuring. The DMERT (Yacobellis, 1983) and DAS operating systems (Goullon, 1978) both provide mechanisms for replacing the individual procedures that comprise a program. Both systems only address the case where the specifications (parameters and return values) of the procedures being updated have not changed and are thus limited to those particular cirumstances. The DYMOS System (Lee, 1983) is a complete dynamic updating system. It provides editors, compilers and a shell to facilitate updating a computer program written in the StarMod language (Cook, 1980). DYMOS will work in a tightly-coupled multiprocessor but does not scale well to a distributed system since it requires a complicated locking protocol for every procedure invocation regardless of whether or not an update is actually being performed.

2. OVERVIEW OF A DYNAMIC PROGRAM UPDATING SYSTEM

This research has adopted replacing components of a program at the level of individual procedures (Frieder, 1989; Segal, 1988). A program is updated by loading the new version of the program and replacing each old procedure with its corresponding new procedure. Various facilities are provided to ensure that the program remains consistent during the update (described in Section 2.4).

2.1. Basic concepts and algorithms

Replacing an old version of a procedure with a new version can be expressed in terms of binding concepts. As a program executes and is updated, the overall function of the program is not changed. Furthermore, the *specification* of the program's procedures (i.e. the operation that each procedure is supposed to perform) is not likely to (but may) change between versions while the procedures' *implementations* (i.e. how each procedure performs the operation) may change. Thus, at any point in time, a procedure's specification is bound to a particular implementation. When a program is updated, the binding of each procedure is changed from its old implementation to its new implementation.

Under the updating system, newer versions of the program can be loaded without affecting the current version's execution. Once the new version has been loaded, the update may be initiated. The update is initiated by a user invoking an update command. The update command interrupts the running program and examines the current state of the run-time stack. Based on this information and the list of all the procedures that each procedure can call (generated by the language compiler), the updating system calculates when each procedure may be updated. Updating a procedure involves changing its binding from the current implementation to the new implementation. When all the procedures have been bound to their new implementation, the program updating is complete.

2.2. Syntactic criteria for updating a procedure

Updating a procedure is based upon the *active* state of the procedure, where *active* is defined as follows:

Let a program Π consist of a set of procedures $P_1,\ldots,P_n$.

Define $\Pi_{rt}(t)$={procedures $P_i \in \Pi$ | P_i's activation record is on the run-time stack at time t}

Let $\delta(P_i)$ denote the dependency function of procedure P_i, which is defined as the set of all procedures that P_i may call.

Define $\delta^*(P_i)$={procedures Q | $Q \in \delta(P_i)$ or $Q \in \delta(\delta^*(P_i))$}

A procedure P_i is *active* iff $P_i \in \Pi_{rt}(t)$ or $\delta^*(P_{i_{new}}) \cap \Pi_{rt}(t) \neq \emptyset$ where $P_{i_{new}}$ is the new version of procedure P_i.

Informally, a procedure P_i is active if and only if it is on the run-time stack or its new version can directly ($\delta(P_{i_{new}})$) or indirectly ($\delta^*(P_{i_{new}})$) call a procedure that is already on the run-time stack. This definition of active procedure will be expanded in Section 2.4 to encompass semantic dependencies as well.

A procedure P that has *not* changed between versions should be updated[2] when the update is first initiated. Alternatively, a procedure Q that has changed between versions may be updated only when it is not active. Since an inactive old procedure cannot become active (as it will have already been converted to a new procedure), examination of the update status need occur only whenever the run-time stack contains less elements than it did when the update was initiated. Thus, checking the size of the run-time stack after the return of each procedure call determines the availability of additional procedures to update.

2.3. Justification for the updating criteria

Given the above definitions, examples of programs where the updating system would not work well can be contrived. Fortunately, if the stated programming requirements are followed, such programs should not occur in many cases. It is assumed that programs are written in a top-down manner (Brooks, 1975; Dahl, 1972; Wirth, 1971). In general, the higher-level procedures, which specify the algorithms of the program, will not change between versions. On the other hand, the lower-level procedures, which describe many of the details used to implement the algorithms, will be more likely to change. The granularity of the procedures (how many statements are in a procedure relative to other procedures) and the depth of the calling tree (the number of calling levels between the 'main' procedure and other procedures) affect updating system performance. If a procedure

[2] In this case, the procedure is not actually replaced but is instead flagged as being 'new' since its old and new versions are the same.

in a program contains many statements and/or is executing much of the time, that procedure will take longer to update.[3]

Also assumed is that data shared by several different procedures, which include global variables, are accessed via abstract data types (ADTs) (Aho, 1983). By centralizing shared-data access, updating the procedure(s) that implement an ADT causes all the procedures that use the ADT to simultaneously get its new version. If a non-updated procedure has made a local copy of some of the ADT's data, they will be correctly mapped to the correct implementation when they are saved in the ADT (Section 2.5).

2.4. Semantic dependencies

The updating criteria described thus far are designed to take *syntactic dependencies* into account. Syntactic dependencies are the relationships between procedures in the program that can be ascertained from the program's syntax. Even if the above coding conventions are followed, it is possible to write programs containing *semantic dependencies*. A semantic dependency is a relationship between procedures that is not detectable from the program's syntax (i.e. two procedures work together to perform some task but do not reference any of the same entities). The updating system deals with semantic dependencies by using information supplied by the programmer. Semantic dependencies are formally stated as follows:

Let $\delta_{up}(P_i)$ = {procedures Q | must be inactive and concurrently updated (if Q has changed) with P_i}.

Let $\delta^*_{up}(P_i) = \{Q \mid Q \in \delta_{up}(P_i) \text{ or } Q \in \delta_{up}(\delta^*_{up}(P_i))\}$.

A procedure P_i can only be updated when it is inactive and all procedures $Q \in \delta^*_{up}(P_i)$ are also inactive. This allows the programmer to specify procedures that must be updated concurrently, thus allowing semantic dependencies to be accommodated. The updating system updates all procedures $Q \in \delta^*_{up}(P_i)$ atomically.

2.5. Maintaining consistency during an update

During the updating process, procedural bindings can be changed such that it is possible for an old procedure to call a new procedure. If the calling sequences and return codes of the procedure in question have not changed between versions, there is no problem. If, however, the new procedure expects different parameters to those that the old procedure is supplying, the parameters must be mapped from the old calling sequence to the new calling sequence. For example, if the old version of procedure `sort` expects an array of integers to sort and the new version expects an array of real numbers, the data must be properly converted before calling the new version of `sort`. Furthermore, because the old program is not aware of the new version of `sort`, any parameter mapping and procedure invocation must be transparent to the old program. In the described system, an *interprocedure* performs this task. An interprocedure is called by the old version of a

[3] In the extreme case, the program consists of a single, large procedure. Under these circumstances, the program cannot be updated since the procedure will be active for the duration of the program's execution.

procedure (with the old parameter format). After the interprocedure maps the parameters with which it is called to the corresponding new format, it invokes the new procedure. Similarly, *mapper procedures* (mprocedures) are used to map local static data within a procedure from the format used in the old version of the program to that used in the new version. Mprocedures can also be used to initialize data structures in the new version of a program based on the state information accumulated in the old version. When used in this manner, mprocedures map state information (but not necessarily data) across versions of the program. For further details, see Frieder (1989) and Segal (1988).

3. INTRODUCTION TO THE PACKET PUMPER

As a sample program that could benefit from a dynamic program updating system, consider the Packet Pumper. At its highest level of abstraction, the Packet Pumper is essentially an internet packet router. The Pumper reads a packet from one of its network interfaces, examines the packet's header to determine its destination address, determines how to get it to the destination, and sends the packet to another network interface. The network interface to which the packet is sent is connected to the network where the packet is destined. Thus, the system 'pumps' packets from several sources to several destinations based on some routeing algorithm. The behaviour of the Packet Pumper is characterized by this routeing algorithm. The updating system is used to alter the system's behaviour by changing the routeing algorithms between versions of the Pumper. Three different versions of the Packet Pumper are presented. The evolution of the Packet Pumper between these versions closely parallels the evolution that has occurred in the United States' Department of Defense (DOD) Internet (ARPAnet) over the past 15 years.

3.4.1. Packet Pumper structure

The fundamental data structure of the Packet Pumper is the *packet*. Every packet consists of a header and zero or more data bytes. All routeing decisions are based on the addressing information found in packet headers. Every header contains a four-byte source and destination addresses for that particular packet. Thus, there can be a total of 2^{32} (about four billion) devices on the network. Dividing this 2^{32}-device space into useful subspaces is the responsibility of the Packet Pumper. Typically, the subspaces approximate the administrative domains which own the networks that the Packet Pumpers interconnect. Routeing is based on the interpretation of the bit fields comprising the 32-bit source and destination addresses.

All three versions of the Packet Pumper use variants of the same high-level algorithm. This algorithm may be expressed as follows:

```
while true do begin
        get_packet (pkt, source_interface);
        rc := route_packet (pkt,dest_interface);
        if rc = PKT_OK then
           put_packet (pkt,dest_interface);
end;
```

The get_packet routine reads the next available packet from a network interface. This packet is passed to route_packet, which places the correct interface of this packet's destination network into dest_interface. Finally, put_packet transmits the packet to this interface.

For illustrative purposes, the details of get_packet and put_packet are not important since they are never updated. The route_packet procedure requires some explanation, however. Route_packet is called with a packet and a destination interface variable. Route_packet examines the packet header and extracts its source and destination addresses. From this information, route_packet selects the network interface that will get the packet closer to its destination. This value is placed into dest_interface and route_packet returns with the value PKT_OK. If route_packet does not know how to route the packet with which it was called, it returns an error value.

3.2. Packet Pumper evolution

Table 1 summarizes the various features of each version of the Packet Pumper.

Version 1 of the Packet Pumper uses a straightforward packet addressing and routeing algorithm: the first byte of the destination address is treated as a network number. The Packet Pumper uses this network number to determine the correct interface to route the packet. One drawback to this addressing scheme is that it partitions the address space into 256 networks of approximately 16 million nodes. Allowing only 256 different networks limits future expandability. Also, relatively few organizations own 16 million devices that are networked together.

To ameliorate this situation, version 2 of the Packet Pumper uses from one to three bytes to designate a network number, with the remaining bytes used to designate a device within a network. In this scheme, the first address byte is divided into three non-overlapping ranges. The first range of values designates a one-byte network address (Class A network), the second range denotes a two-byte network address (Class B network), and the third range denotes three-byte network addresses (Class C network). This addressing scheme allows the network address space to be tailored to different types of networking environments with varying numbers of computers. Another advantage of this approach is that the owner of a Class A network can logically partition it into a number of Class B or C subnetworks. One disadvantage to this approach is that it does not allow dynamic rerouteing of packets due to transient conditions on the network (overloaded Packet Pumpers or Pumper downtime).

Table 1. Summary of Packet Pumper features

Version	Main addressing features	Problems
1	One byte net address, three byte node address	Too few networks each containing too many nodes
2	One, two or three byte network addresses with variable numbers of nodes per network	Does not allow dynamic rerouteing of packets due to congestion or Pumper downtime
3	Dynamic routeing and default routes	Must support both control and data packets

Version 3 of the Packet Pumper allows special control packets to be passed to other Packet Pumpers. These control packets instruct a Packet Pumper to use a different routeing algorithm. This allows quick adaption to changing network conditions. Routes may be both added and deleted. Version 3 also supports the concept of *default routes*. A default route is a network to which a Packet Pumper sends a packet if it does not know how to get a packet to its destination. As with other dynamic routes, a default route can be added or deleted when necessary. For this version to function properly, the Packet Pumper must be modified to process both control and data packets.[4]

3.3. Code required to update the Packet Pumper

In a simplified implementation of versions 1 and 2 of the Packet Pumper, the route_packet procedure is coded as a set of **if** statements as shown below.

```
function route_packet (Packet_t pkt, integer inter) returns integer;
begin
    net_number := get_net_number (pkt);

    if net_number < 17 then begin
        inter := 1;
        return (PKT_OK);
    end

    if net_number >= 17 and net_number < 43 then
    begin
        inter := 2;
        return (PKT_OK);
    end

    if net_number >= 43 and net_number < 159 then
    begin
        inter := 3;
        return (PKT_OK);
    end

    if net_number >= 159 then
        return (NETWORK_UNREACHABLE);
end;
```

Since there are a small number interfaces per Packet Pumper (three in this example), the code shown above is a simple and effective way of performing the routeing. To calculate a network number for each packet, route_packet calls the get_net_number

[4] We assume that there is a field reserved in the packet header for 'future expansion', 'packet type', or 'packet version'. Good design dictates that space be reserved for future use/system expansion. For example, most modern CPU architectures reserve opcodes, addresses, and other objects for the 'future use' of the CPU designers This was also done in the ARPAnet. ™SunOS is a trademark of Sun Microsystems, Inc. ™Unix is a trademark of AT&T Bell Laboratories.

procedure. When we update from version 1 to version 2, the only procedures that must be replaced are route_packet and get_net_number. Since neither of these procedures' calling conventions change between versions, nor do they use any internal static data, no interprocedures or mprocedures are required for this update. In version 2, get_net_number will be modified to calculate the correct (version 2) network number and additional **if** statements will be added to the route_packet code to allow packets to be routed to the additional networks.

Updating from version 2 to version 3 requires non-trivial changes to the Packet Pumper code. The code that implements the algorithm given in Section 3.1 must be modified to account for both control and data. This revised algorithm is shown below.

```
while true do begin
        get_packet (pkt,source_interface);
        case pkt.type of
                pkt_t_data:      do_route_xmit (pkt,
                                         source_interface);

                pkt_t_ add_rt:  do_route_add(pkt);

                pkt_t_del_rt:    do_route_del (pkt);
        end;
end;
```

The algorithm examines the type of each packet and dispatches the packet to an appropriate routine for further processing. Each of the do_route_*xxx* procedures shown above eventually calls route_packet. Route_packet has been modified to accommodate dynamic routeing. The version 3 route_packet manages an internal routeing table data structure which is used to make routeing decisions. As a result of this increased functionality route_packet now takes an additional parameter. The new parameter tells route_packet whether it is calculating a route or performing routeing table maintenance.

To maintain consistency between version 2 and version 3, an interprocedure and an mprocedure are required. The interprocedure maps route_packet calls made by version 2 procedures to the correct calling conventions of the version 3 route_packet. The interprocedure's code looks like:

```
interprocedure route_packet (Packet_t pkt,
                              integer interface)
                                       returns integer;
begin
        command := ROUTE_FIND;

        switch_ calling_domain_to_version_3;
        rc := route_packet(pkt,interface,command);
        switch_calling_domain_to_version_2;
        return (rc);
end;
```

This interprocedure loads the third parameter of the version 3 route_packet with a value that makes sense in the version 2 domain, calls the operating system/updating system to call the version 3 routine from the version 2 interprocedure, performs the call, resets the calling domain, and returns the route_packet return code. This interprocedure correctly maintains the version 2 route_packet semantics in the domain of version 3.

When the version 3 Packet Pumper starts up, no packets can be routed since the routeing table is initially empty. If starting from scratch, this would be fine but version 2 already has a hard-coded routeing table. To make the update transparent, the version 2 routeing table must somehow be coded into version 3 before the version 3 route_packet is executed. Achieving this transparency can be accomplished using an mprocedure. This mprocedure merely calls the version 3 route_packet routine in specifying the 'add-route' operation. This is done once per route coded in version 2. When all version 2 routes have been added to the version 3 route_packet, version 2 procedures calling route_packet via the interprocedure will behave correctly.

3.4. Packet Pumper summary

An overview of the behaviour of each version of the Packet Pumper has been provided. The evolution of each version to the subsequent version represents a logical progression to increasing complexity and functionality. Although the actual program is simplified, the logical migration from one version to the next is the same as has occurred in the Internet. Thus, this example represents a realistic scenario of program evolution. Furthermore, shutting down a single packet router impacts connectivity and performance across the entire network. Hence, having the ability to update packet router software without shutting down the packet router is an important step in increasing network availability and reliability.

4. UPDATING THE PACKET PUMPER

Prior to discussing the Packet Pumper directly, a brief overview of the prototype is provided. For further details, see Frieder (1989).

4.1. The prototype updating system

A prototype updating system has been constructed to validate and benchmark the updating system architecture. The prototype executes on Sun Microsystems computers running SunOS™ (Sun, 1986) (a BSD 4·3-compatible Unix™ system) and consists of several major components. The primary user interface to the prototype is called the Updating Shell (*ush* or 'u-shell'). The ush reads commands typed from the user's terminal and, based on the type of command, performs some local action or interacts with other components of the updating system if necessary. The ush is capable of dynamically loading and linking[5] user programs to be updated.

[5] This feature is not available in standard BSD Unix. It was implemented in the prototype using the standard Unix linker (ld) and our own loading system.

User programs are not executed in the same address space as the ush, but rather in the address space of a separate component of the updating system called the Program Update Processor (*pup*). When a user program is loaded by the ush, it is downloaded into the pup. The pup and the ush communicate with each other via internet domain sockets (Sun, 1985), and thus need not reside on the same physical computer system. The pup and ush do not interact directly with the sockets but through a communication abstraction reminiscent of remote procedure calls. This communications subsystem allows messages representing commands to the pup and ush to be interchanged without regard to the idiosyncrasies of sockets. Block data transfers as well as asynchronous message notification are also supported. TCP port management and ush/pup connection control are also provided but are not discussed here.

Table 2 lists the commands available in the ush. These commands are demonstrated in Section 4.2.

4.2. Updating the Packet Pumper

When the ush is started, the following is displayed:

```
citi% ush
Updating System Shell version 0.0 (compiled on Sep 26/88 at 01:04)
      running under SunOS version 4.0 on host citi

>
```

Table 2. Ush commands

Command	Function
com[puter?]	make sure the computer still works
con[nect]	connect to a pup
echo	echo message on pup and ush
help	repeat this display
help topic	detailed help on 'topic'
host	select name of current host
load	link and load a version of a program
proc[edures]	display version information on all procedures
procs	display version information on all procedures
prog[ram]	select name of current program
qu[it]	end updating shell session
run	run a loaded program
so[urce]	read and execute commands from a file
stat[us]	display status of current pup
up[date]	begin an update operation
uncon[nect]	break connection with a pup
ctrl-D	end updating shell session
! cmd	shell escape

When the pup is started, the following is displayed:

```
citi% pup x
program update processor version 0.0 (compiled on Sep 26/88 at
01:04)
          running under SunOS version 4.0 for program x
```

To connect the shell started above to the pup, the ush must be given a symbolic name of a program and told which host on the network is running a pup for the desired program. The program name corresponds to the name of the program supplied on the pup (x in the example above). If no host is specified in the connect command, the ush assumes that the pup resides on the same host as the ush. The ush and the pup need not reside on the same physical comptuer.

```
> prog x
The current program is now x
> con
Connected to pup for program x on host citi
Pup for program x on host citi is new
>
```

Once the connection has been established, programs may be loaded and run. To load version 1 of the Packet Pumper, the following is performed:

```
> load 1 pkt1.delta pkt1.o
Load sequence begins. . .
   extracted updatable procedures from delta file pkt1.delta
   program x has 7 updatable procedures
   program x's procedures registered with pup name service on
      host citi
   delta* computed for 7 procedures in x:

         0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
         0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
         0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
         0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                      .
                      .
                      .
   program object code image requires 15824 bytes
   program loading at address 0x26a18 in pup x's address space
      on host citi
   program linking complete
   loaded VMC segment descriptor table into citi's VMC
```

```
        copying code to pup ................ complete
        no interprocedures loaded for version 1
        no mprocedures loaded for version 1
    Version 1 of program x loaded successfully
    >
```

This display is printed with full debugging compiled into the ush. The δ table (Section 2.2) printout has been shrunk for the purpose of this example.

Once the program has been loaded, it may be run immediately or additional versions of the Packet Pumper can be loaded. Loading version 2 now results in:

```
> load 2 pkt2.delta pkt2.o
Load sequence begins . . .
   extracted updatable procedures from delta file pkt2.delta
   program x has 7 updatable procedures
   program x's procedures registered with pup name service on
      host citi
   delta* computed for 7 procedures in x:
         delta table printout eliminated for clarity
   program object code image requires 15944 bytes
   program loading at address 0x2a7f8 in pup x's address space
      on host citi
   program linking complete
   loaded VMC segment descriptor table into citi's VMC
   copying code to pup ................ complete
   iproc object code image requires 560 bytes
   iproc loading at address 0x2e648 in pup x's address space on
      host citi
   iproc linking complete
   copying code to pup . complete
   mproc object code image requires 144 bytes
   mproc loading at address 0x2e888 in pup x's address space on
      host citi
   mproc linking complete
   copying code to pup . complete
Version 2 of program x loaded successfully
Version 1 interprocedures loaded successfully
Version 1 mprocedures loaded successfully
>
```

Alternatively, a new version of the program while the current version is executing can be loaded. Suppose instead that version 1 of the program was run. This is accomplished by:

```
> run
Program x is now running
>
```

When the ush is given the run command, version 1 of the current program is executed. On the pup, we see the following if debugging is enabled:

```
pup: INTERRUPT! (pup state: 0) received command RUN
Execution begins at address 0x26a18 . . .
packet: type 0x01 addr [0x0a0b0c0d,0x04020304] intf [0x00,0x01]     0
packet: type 0x01 addr [0x12345678,0×9abcdef0] intf [0x09,0x03]     0
packet: type 0x01 addr [0x22334478,0x9a01def0] intf [0x09,0x03]    16
packet: type 0x01 addr [0x22334478,0x9867def0] intf [0x09,BAD*]    16
packet: type 0x01 addr [0x01001234,0x02000091] intf [0x02,0x01]    16
packet: type 0x01 addr [0x12345678,0x23bcdef0] intf [0x03,0x00]    16
packet: type 0x02 addr [0x01001234,0x02000091] intf [0x02,0x01]    16
```

The INTERRUPT message signifies that the pup has received a RUN command from the ush. This causes version 1 of the currently loaded program to begin executing. The messages prefaced by packet: are output from the Packet Pumper used for debugging and performance analysis. These messages are read as follows. The type is the packet type. The bracketed addr pair is the hexadecimal source and destination network address of the packet being routed. The bracketed intf pair represents the source and destination network interface numbers of the packet being routed. If there is no route to a given network, BAD* is placed in the destination network interface field. Finally, the number at the end of the line represents the amount of task CPU time that has been expended by the Packet Pumper. This is used for performance analysis.

For example, the first packet of this screen dump is of type 1 (data packet)[6] and came from network address Ox0a0b0c0d and is destined for network address 0x04020304. The packet arrived at the Packet Pumper on network interface 0 and was sent out on network interface 1. The Packet Pumper continues to read and pump packets in cycles. Source address 0x0a0b0c0d signifies the beginning of the cycle of packets. The cyle repeats to demonstrate the behaviour of the system in different versions of the Pumper.

After some period of time, suppose an update of the Packet Pumper to version 2 was desired. On the ush side:

```
> update pkt2.changes
Update of program x from version 1 to version 2 begins . . .
  2 procedures have changed between versions:
        _route_packet
```

[6] In version 1 and 2 of the Packet Pumper, the type field is unused and ignored. Version 3 interprets the type field as a routeing command.

```
                  _get_net_number
            program x runtime stack backtrace:
                  _do_route
                  _main2
            warped 5 unchanged procedures into version space 2
            computed inactive procedures:
                  _route_packet
                  _get_net_number
            warped 2 changed procedures into version space 2
          Program x updated to version 2 successfully
          >
```

The parameter pkt2.changes is a filename which contains the list of procedures that have changed between versions. This information was prepared by the programmer who wrote version 2 of the Packet Pumper. After the update command is given, the results on the pup side (with debugging enabled) are shown below:

```
packet: type 0x01 addr [0x12345678,0x9abcdef0] intf [0x09,0x03] 320
packet: type 0x01 addr [0x22334478,0x9a01def0] intf [0x09,0x03] 320
                        .
                        .
                        .
packet: type 0x01 addr [0x01001234,0x02000091] intf [0x02,0x01] 480
packet: type 0x03 addr [0x01001234,0x02000091] intf [0x02,0x01] 480
pup: INTERRUPT! (pup state: 2) received command UPDATE
    pup: UPDATE dispatcher received command GET_RTS
    pup: UPDATE dispatcher received command SET_SDTE
    pup: UPDATe dispatcher received command SET_SDTE
    pup: UPDATE dispatcher received command SET_SDTE
    pup: UPDATE dispatcher received command SET_SDTE
    pup: UPDATE dispatcher received command SET_SDTE
    pup: UPDATE dispatcher received command SET_SDTE
    pup: UPDATE dispatcher received command SET_SDTE
    pup: UPDATE dispatcher received command SET_HRVN
    pup: UPDATE dispatcher received command EOF
packet: type 0x03 addr [0x01001234,0x02000091] intf [0x02,BAD*] 500
packet: type 0x01 addr [0x22334478,0x9867def0] intf [0x09,BAD*] 500
                        .
                        .
                        .
packet: type 0x01 addr [0x12345678,0x9abcdef0] intf [0x09,0x04]  560
packet: type 0x01 addr [0x22334478,0x9a01def0] intf [0x09,0x03]  580
```

The lines preceded with pup: are the debugging output of the pup interacting with the ush to perform the update. They correspond to the ush's view of the update operation. The GET_RTS command sends a snapshot of the Pumper's run-time stack to the ush, the SET_SDTE command manipulates the segment descriptor tables of the pup's virtual memory controller. This is how procedures are bound into the new version's space. Finally, SET_HRVN sets the pup's notion of the highest running version number. Due to the simplicity of this update, it could complete in a single pass.

After the update is completed, some packets that may have been routeable before may have different destinations (or none at all) due to the fact we are now using a different algorithm. For example, the first two and last two packets on the above screen dump are sent to the same interface before the update, and to different interfaces afterwards. Looking closely at the network numbers of the packets' destinations, we see that before the update, both packets were bound for network number 0x9a. After the update, however, the first packet is sent to network 0x9abc while the second packet is sent to network 0x9a01. The destination network interface numbers are indeed sending the packets to (the correct) different network interfaces.

To update version 2 to version 3 of Packet Pumper, assuming version 3 is already loaded the following command is typed:

```
> update pkt3.changes
Update of program x from version 2 to version 3 begins . . .
   2 procedures have changed between versions:
         _do_route
         _route_packet
   program x runtime stack backtrace:
         _do_route
         _main2
   warped 5 unchanged procedures into version space 3
   computed inactive procedures:
         _route_packet
   warped 1 changed procedures into version space 3
   enabled procedure return interrupts
Program x update initiated successfully
>
         some time passes . . .
   ***Program x updated to version 3 successfully***
```

This time the update behaved differently than before. Because there were active procedures, the update could not complete when initiated. Since a procedure is only updated when it is *not* active, the updating of procedure do_route is delayed. Until *all* of the procedures have been updated, each time a procedure invocation terminates, the ush is notified. The ush determines if active procedures are now inactive and, if so, updates them. This is done asynchronously within the ush and is normally undetectable to the user. After all procedures have been updated, the pup asynchronously notifies the ush, which outputs a message similar to the one shown above.

The update shown above (on the pup side):

```
packet: type 0x01 addr [0x22334478,0x9867def0] intf [0x09,BAD*]   740
packet: type 0x01 addr [0x01001234,0x02000091] intf [0x02,BAD*]   740
packet: type 0x01 addr [0x12345678,0x23bcdef0] intf [0x03,0x00]   740
   pup: INTERRUPT! (pup state: 2) received command UPDATE
```

```
        pup: UPDATE dispatcher received command GET_RTS
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_HRVN
        pup: UPDATE dispatcher received command EI_POP
        pup: UPDATE dispatcher received command EOF
    packet: type 0x02 addr [0x01001234,0x02000091] intf [0x02,BAD*]    760
        pup: PROCEDURE INVOCATION TERMINATED!
        pup: UPDATE dispatcher received command GET_RTS
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command DI_POP
        pup: UPDATE dispatcher received command EOF
    route  : type 0x02 route network 0x9867    via intf 0x08 added     800
    route  : type 0x02 route network 0x0       via intf 0x06 added     800
    packet: type 0x01 addr [0x22334478,0x9867def0] intf [0x09,0x08]    800
    packet: type 0x01 addr [0x01001234,0x02000091] intf [0x02,0x06]    800
    route  : type 0x02 route network 0x2       via intf 0x07 added     800
    packet: type 0x01 addr [0x01001234,0x02000091] intf [0x02,0x07]    800
    route  : type 0x03 route to network 0x2        deleted             800
   packet: type 0x01 addr [0x01001234,0x02000091] intf [0x02,0x06]     800
    route  : type 0x03 route to network 0x2        nonexistent         800
    packet: type 0x01 addr [0x22334478,0x9867def0] intf [0x09,0x08]    800
    packet: type 0x01 addr [0x22334478,0x0067def0] intf [0x04,0x06]    820
    route  : type 0x03 route to network 0x9867     deleted             820
    packet: type 0x01 addr [0x22334478,0x9867def0] intf [0x09,0x06]    840
    packet: type 0x01 addr [0x12344129,0x99294017] intf [0x07,0x06]    840
    route  : type 0x03 route to network 0x0        deleted             860
    packet: type 0x01 addr [0x22334478,0x0067def0] intf [0x04,BAD]     860
    packet: type 0x01 addr [0x12344129,0x99294017] intf [0x07,BAD*]    880
    route  : type 0x03 route to network 0x0        nonexistent         880
    packet: type 0x01 addr [0x22334478,0x0067def0] intf [0x04,BAD*]    880
```

The first block of pup updating commands looks similar to the first update with one exception. Before returning control to the Packet Pumper, the pup executes the EI_POP command. This causes the pup to enable *interrupts* each time a procedure is popped from the run-time stack. This continues until the update is finished. After the first block of updating commands has completed, the packet that was being processed is completed and is printed out. Notice that because this packet was processed before the new version of the Pumper was installed, it is still interpreting the packet by version 2 rules (i.e. ignoring packet type). When this packet has been printed out, the pup takes a procedure termination interrupt. In this case, the do_route procedure has terminated and we continue with the update. After do_route has been updated, the pup executes the DI_POP command, which disables the procedure pop interrupts. The update is now complete.

On the very next packet, the new version is running. The packet is a route add command (type 2) and a new route is successfully added. After this has been performed, another route (the default route) is added and two packets that were unsuccessfully pumped before the update can now be successfully routed since the packet pumper now knows (courtesy of the route add packets and dynamic routeing) how to do it. As can be seen later in the output shown above, routes can also be deleted and, as expected, packets can no longer be routed to non-reachable networks.

5. PERFORMANCE OF THE UPDATING SYSTEM

To measure the performance penalty during an update, the prototype updating system and the Packet Pumper were instrumented to produce timing information that can be analysed with other tools. The pup contains a number of performance meters which may be read and written. One of these meters keeps track of the amount of task CPU time that the pup (and loaded programs) expend. Internally, this time is based on the Unix task CPU time of the pup process. By observing the amount of useful work performed (i.e. number of packets routed) over time, we can obtain a reasonable approximation of the Packet Pumper's overall performance. By examining the time interval(s) where the update(s) occur, we can calculate the cost of performing the update.

The precision of the CPU time meter is a direct function of the precision of the internal Unix process timing routines. The prototype updating system currently runs on Sun 3 and Sun 4 systems running SunOS 4·0. Both Sun systems time processes in units called 'ticks'. On the Sun 3, one 'tick' equals 1/50 second of CPU time, while on the Sun 4, one 'tick' equals 1/100 second. Thus, the CPU use can be measured to the nearest 10–20 ms of task CPU time. Although this granularity is somewhat coarse, it is sufficient to obtain prototype performance data.

To approximate the performance of each version of the Packet Pumper before performing any updates, each version ran several times on a lightly loaded Sun 3/280 computer. Based on 15 runs of the same data set for each version, version 1 of the Packet Pumper pumps at 7·26 pkts/CPU tick, version 2 at 7·67 pkts/CPU tick, and version 3 at 5·40 pkts/CPU tick. The performance of versions 1 and 2 are roughly comparable while version 3 runs somewhat slower due to the increased complexity of its routeing algorithm.

To determine the performance degradation during a complete update, all three versions of the Packet Pumper were loaded and ran version 1. At a random time an update to version 2, and later to version 3 was issued. This test was performed 5 times. As stated earlier, the update from version 1 to version 2 is straightforward; only one procedure changed between versions. In all runs of the test, this update occurred in no more than one clock tick (i.e. no more than 20ms on the Sun 3/280) but because the program was updated at different points in its execution, there is some variation[7] in the exact time between runs.

Updating from version 2 to version 3 is more complex due to the state information that must be retained from version 2. Recall that version 3 maintains the routeing information (that was previously represented by the version 2 routeing code) in a table

[7] The Unix clock also contributes to this variation somewhat. Auxiliary experiments showed that this effect was not significant when the Sun 3/280 was idle.

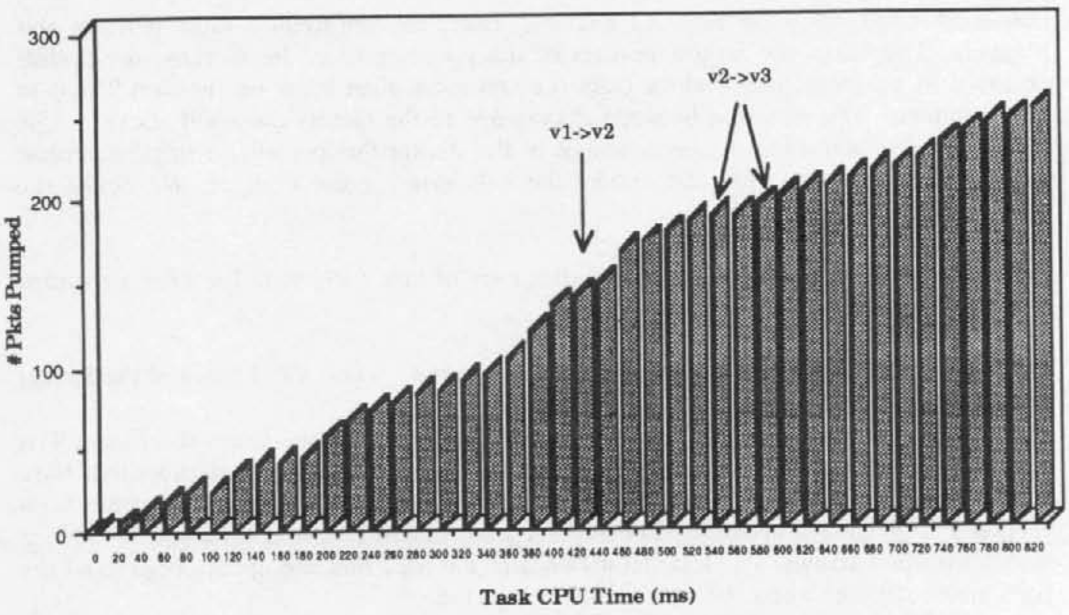

Figure 1. Number of packets pumped over time

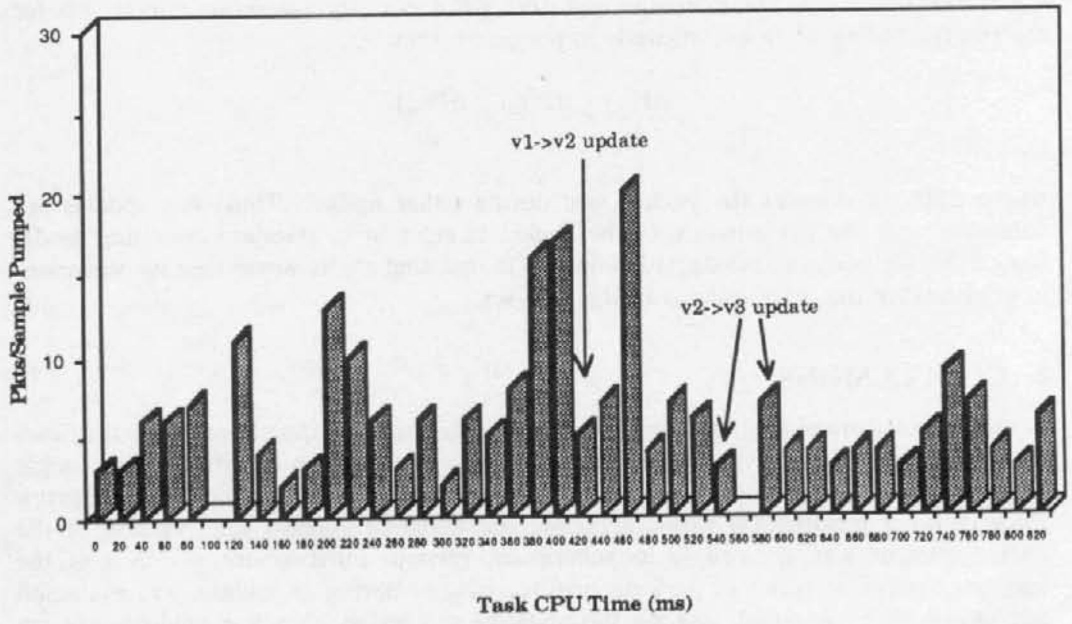

Figure 2. Packet rate over time

that is changed based on routeing packets. Thus, an mprocedure must perform this mapping. This takes the largest portion of the updating time. In all runs, the update occurred in no more than 4 clock ticks (i.e. no more than 80ms on the Sun 3/280) to fully complete. The variation between runs is due to the factors discussed above.

Two graphs illustrating the performance of the Packet Pumper over a single complete run are shown below. This data model the behaviour of the Pumper. We define the Pumping Function $P(t)$:

$P(t)$ = the number of packets pumped after t ms of task CPU time has been expended by the Packet Pumper (and pup)

Figure 1 shows $P(t)$ plotted over the execution lifetime (in task CPU time) of the Packet Pumper.

Because the timing data is based on discrete samples at 20 ms intervals, Figure 1 is plotted as a bar graph. At each 20 ms interval, the total number of packets that have been pumped thus far is shown. The v1->v2 notation denotes when the update from version 1 to version 2 occurred. The version 2 to version 3 update (denoted v2->v3) is shown with two arrows. The left arrow indicates the time that the update began and the right arrow denotes when the update was completed.

A related measure of Packet Pumper performance is how the pumping *rate* changes over this time interval. Mathematically, this relationship may be expressed as $\mathrm{d}P(t)/\mathrm{d}t$. Since the time intervals are discrete quantities, $\mathrm{d}P(t)/\mathrm{d}t$ may be approximated as the number of packets pumped during a given time interval. This is shown below in Figure 2 for the same Packet Pumper run as in Figure 1.

Let $\mathrm{d}P(t_l)/\mathrm{d}t$ denote the minimum and $\mathrm{d}P(t_h)/\mathrm{d}t$ denote the maximum packet rate for the run (excluding when an update is in progress), then

$$\frac{\mathrm{d}P(t_l)}{\mathrm{d}t} \leqslant \frac{\mathrm{d}P(t_u)}{\mathrm{d}t} \leqslant \frac{\mathrm{d}P(t_h)}{\mathrm{d}t}$$

where $\mathrm{d}P(t_u)/\mathrm{d}t$ denotes the packet rate during either update. Thus, the updates are consistent with the performance of the Packet Pumper in its standard operating mode. Notice that the performance degradations due to updating are no worse then the variations in performance due to variations in the data set.

6. CONCLUSIONS

A method of dynamically updating computer programs, an example program that uses the system, and a prototype version of the updating system are described. The sample program, the Packet Pumper, makes packet routeing decisions for a computer network gateway for a hypothetical network. Using the updating system, each version of the Packet Pumper was updated to its subsequent version. Furthermore, the cost of the updating system, in terms of performance degradation during an update, was evaluated and shown to be minimal, making the updating system an attractive environment for applications that must be continuously running. As computer systems become more complex, methods of reducing software downtime during software maintenance will become increasingly important.

Acknowledgements

We wish to thank Bruce Schatz and the other reviewers for both their technical and stylistic comments. We especially wish to thank Mel Colter for his invaluable assistance. His help is greatly appreciated. This research is supported by the University of Michigan Center for Information Technology Integration, 2901 Hubbard Ave., Ann Arbor, MI 48105, U.S.A.

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1983) *Data Structures and Algorithms*, Addison-Wesley Publishing Company, Reading, Mass.

Birrell, A. and Nelson, B. (1984) 'Implementing remote procedure calls', *ACM Transactions on Computer Systems*, **2**(1), 39–59.

Bloom, T. (1983) *Dynamic Module Replacement in a Distributed Programming System*, PhD Dissertation, MIT.

Bodwin, J. (1987) Personal communication regarding the internals of the Michigan Terminal System.

Brooks, F. P., Jr. (1975) *The Mythical Man-Month*, Addison-Wesley Publishing Company, Reading, Mass.

Cook, R. P. (1980) 'StarMod—a language for distributed programming, *IEEE Trans. Software Engineering*, SE-**6**(6), 563–571.

Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R. (1972) *Structured Programming*, Academic Press, London and New York.

Fabry, R. (1976) *How to Design a System in which Modules can be Changed on the Fly*, in Proc. Second International Conference on Software Engineering, IEEE, October, pp. 470–476.

Frieder, O. and Segal, M. (1989) 'On dynamically updating a computer program: from concept to prototype', *Journal of Systems and Software*, under revision.

Goullon, H., Isle, R. and Löhr, K. (1978) 'Dynamic restructuring in an experimental operating system', *IEEE Trans. Software Engineering*, SE-**4**(4), 298–307.

Lee, I. (1983) *DYMOS: A Dynamic Modification System*, PhD Dissertation, University of Wisconsin.

Rey, R. F. (ed) (1986) *Engineering and Operations in the Bell System* (second edition), AT&T Bell Laboratories, Murray Hill, NJ.

Schell, R. (1971) 'Dynamic reconfiguration in a modular computer system', Tech. Rep. TR-86, MIT Laboratory for Computer Science.

Segal, M. and Frieder, O. (1988) *Dynamic Program Updating in a Distributed Computer System* in Proc. IEEE Conference on Software Maintenance, October, pp. 198–203.

Sun (1985) 'Interprocess communication primer', *Networking on the Sun Workstation*, Sun Microsystems, Inc., Mountain View, CA, April.

Sun (1986) *Unix Interface Reference Manual*, Sun Microsystems, Inc., Mountain View, CA.

Wirth, N. (1971) 'Program development by stepwise refinement', *CACM* **14**(4), 221–227.

Yacobellis, R. H., Miller, J. H., Niedfeldt, B. G. and Weber, S. S. (1983) 'The 3B20D processor & DMERT operating system: field administration subsystem', *The Bell System Technical Journal*, **62**(1), 323–339.