Multiple Bypass: Interposition Agents for Distributed Computing (Preprint Version)

Douglas Thain and Miron Livny

University of Wisconsin
Computer Sciences Department
1210 W. Dayton St. Madison WI 53703
{thain,miron}@cs.wisc.edu

Interposition agents are a well-known device for attaching legacy applications to distributed systems. However, agents are difficult to build and are often large, monolithic pieces of software which are suited only to limited applications or systems. We solve this problem with Bypass, a language and a tool for quickly building multiple small agents that can be combined together to create powerful yet manageable software.

1. Introduction

A wide variety of distributed systems can send a user's applications to computers spread around the world. However, an application may not be able to run correctly on every system that it can access.

The system may not provide the interface that that the program expects. For example, metacomputing systems such as Globus [6] and Legion [8] provide extensive capabilities but generally require applications to be re-written to take advantage of their systems. An application written to a standard interface, such as POSIX [10], cannot make full use of these facilities.

The system might provide the correct interface, but might not have the correct resources. For example, a matchmaking system such as Condor [14] might match an application in North America to a compatible computer in Europe. Although the application and the system use the same interface, the system is not likely to have the application's needed data files or even have a userid with which to represent the remote user.

Such problems are solved in Condor and other systems by an *interposition agent* [12], or *agent* for short. An agent can place itself between an application and the operating system and trap some of its procedure calls. The agent might transform the application's calls to a new interface, or it might find an indirect way of computing the correct results. A common technique is to send trapped calls to a *shadow process* on the user's home machine, which can compute the correct results based on the user's home environment. A system involving an application, agent, and shadow is known as a *split execution* system and is shown in figure 1.

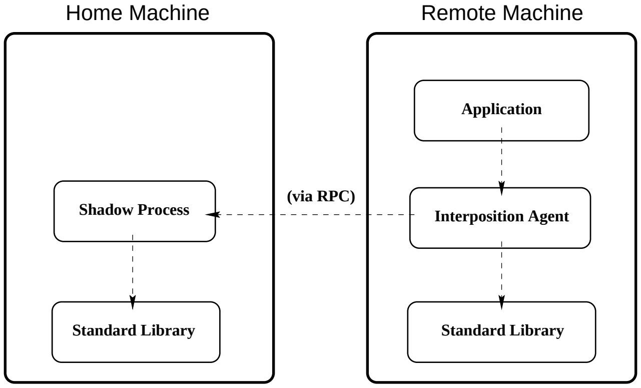

Figure 1. Split Execution

The design of split execution systems is an open research issue. For example, data may be lazily or aggressively cached between an agent and the shadow. Policy decisions regarding call routing may be implemented at the shadow, the agent, or explicitly within the user program. Both the agent and the shadow may be given complex mechanisms for servicing an application's system calls.

The construction of the agents themselves is hard. The mechanism for implementing an agent is sensitive to the domain of applications to be used. Current agents often combine unrelated facilities, yielding overly restrictive systems. Agents are difficult to specify in a portable manner. Semantics for combining agents are not well defined.

In this paper, we present Bypass, a tool for quickly building split execution systems. Bypass does not implement one particular system, but is a framework upon which many different split execution systems can be built. Bypass is also capable of building standalone agents that may be combined together to form more complex systems.

We have several contributions to previous work. We present a simple language for specifying agents and shadows independent of implementation mechanisms. We use an interposition mechanism that is suitable to our problem domain – distributed systems. We propose rules governing the composition of multiple agents. Finally, we illustrate and justify our rules using practical examples.

This is an extended version of an earlier publication [2]. In this paper, we repeat the introductory material, but give more extensive examples and describe an improved mechanism for layering multiple agents. Our measurements use the same methodology, but are repeated on the new software, yielding slightly different results.

2. Goals

We envison a system where the programmer writes a simple specification of a split execution system merely in terms of the calls to be trapped and the action to be taken. The system should hide all of the ugly implementation details and be able to produce code for a wide variety of systems. Our goals for this system are:

1. *Allow splitting of unmodified applications.* There are a wide variety of applications already written for the POSIX interface. Very few users are willing to rewrite their applications to take advantage of specialized distributed computing interfaces: they may be unwilling to invest valuable time in exchange for unknown benefits, they may be unable to modify a commercial application, or they may simply not have the knowledge to rewrite an application. Tools must work with existing, untouched, executable programs.

2. *Allow dissimilar systems to interact.* In order to harness the maximum number of worker machines for a large distributed computation, one must be willing and able to harness machines of varying architectures and operating systems. Tools must form a translating layer that allows inter-operation between software components on dissimilar machines.

3. *Separate the programmer's intent from the necessary mechanism.* We will show that trapping procedure calls involves knowledge of unpleasant implementation details. The programmer is not interested in creating or dealing with this knowledge for every new program. Furthermore, there are multiple ways of trapping procedure calls, each with their own benefits and drawbacks. A tool should not require the programmer to mix implementation details with the specification of an agent.

4. *Incur minimal overhead.* We expect that this tool will allow the programmer to attach a variety of (possibly slow) mechanisms to a program. However, the trapping mechanism itself should not have undue overhead.

3. Difficulties

Speaking from the experience of developing the Condor system, we assert that hand-coding split execution systems is *hard.* Trapping a few system calls on one particular operating system is easy, but trapping all of the system calls, passing them between dissimilar machines, and porting the software to a wide variety of platforms involves coming to terms with the following difficulties:

1. *Obscured interfaces.* For example, the POSIX `stat` system call returns summary information about a file. The structure returned by `stat` has changed as architectures have moved from 16 to 32 to 64 bits. As a result, the `stat` defined in most standard libraries assumes an obsolete definition of the structure. Recent programs that appear to use `stat` at the source level are actually redirected, by way of a macro or inline function, to a system call often named `fxstat`. A program intending to trap `stat` must actually trap several different entry points and manage several different structures.

2. *Varied implementations.* For example, `socket` is a well-known library interface for creating a communication channel. However, several systems do not implement `socket` by invoking a matching `socket` system call. Some systems implement it as `open` on a special file, followed by an `ioctl`. Others implement it as a call to `so_socket`, whose additional arguments and semantics are undocumented. A program indending to trap `socket` and pass it between different systems must be able to trap the actual procedure call, and not the corresponding system calls, whatever they may be.

3. *Binary incompatibilities.* For example, most varieties of UNIX conform to source-level standards such as POSIX. These standards require that certain types, symbols, and structure elements be defined at the C source level, but do not specify implementation details such as the number of bytes in a type, the actual value assigned to a symbol, the concrete types expected by an interface, or the number and ordering of elements in a structure. Figure 2 lists examples of these binary differences on three platforms supported by Condor.

	OSF/1 4.0 Alpha	Linux 2.2 Intel	Solaris 2.6 Intel
Size of `off_t`	8 bytes	4 bytes	4 bytes

Arguments to `send`	`int,`
	`void *,`
	`unsigned,`
	`int`
		`int,`
		`void *,`
		`int,`
		`unsigned`
			`int,`
			`void *,`
			`unsigned,`
			`int`

Value of `O_CREAT`	0x200	0x040	0x100
Elements in `struct utsname`	5	6	5

Figure 2. Binary Incompatibilities

4. Bypass

Bypass is our first implementation of these ideas. Bypass reads a platform-independent specification file and produces source code for an agent and a shadow. The agent is compiled into a dynamic library and the shadow is compiled as a standalone executable. The agent can be easily linked into an existing application at run-time, yielding a program prepared for split execution. Bypass hides many of the difficulties described above, allowing the programmer to concentrate on larger issues.

4.1. Language

The language accepted by Bypass is very similar to C: it is simply a list of procedure declarations. Each procedure to be trapped is named along with its parameters and types, followed by *two* code blocks: one for the code to be executed at the agent, and one for the code to be executed at the shadow. Any necessary headers, helper functions, or variables may be placed in a prologue above the declarations.

Pointers in C are ambiguous, so pointer arguments must be annotated with keywords that cement their meaning. This is necessary if the implementation of the agent is such that a callee is executed in a different address space than the caller. Currently, this is only used when calls are sent to a shadow process for execution.

When an application invokes a trapped procedure, the code block of the named procedure in the agent is executed. This block may contain any arbitrary C code, including calls to the trapped procedures themselves. Such calls appear to be recursive, but in fact refer to the original definition. The precise rules for binding names to procedures are a little tricky and are described below in section 4.2.

When building agents, it is typical, but not required, to invoke the original procedure within the agent action. An action may also invoke other trapped procedure calls, or it may compute a value and return without invoking any other procedures at all.

Instead of giving an exhaustive definition of the Bypass language, we will illustrate it with three examples. These examples are complete agents that may be compiled and used with real programs. Although they are not very complex by themselves, they may be combined into a meaningful remote execution system.

We will assume the agents are all applied to an application shown in figure 3. This application makes use of a number of POSIX functions found in a standard library. Each reference to a procedure name normally invokes the definition found in the standard library. This binding is indicated by an arrow running from the application to a definition in the library. As we add agents, these bindings will change.

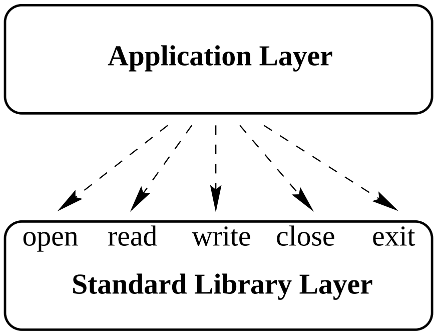

Figure 3. Unmodified Application

```
ssize_t read( int fd,
              out opaque "length" void *data,
              size_t length )
agent_action
{{
    if( fd<3 ) {
        return bypass_shadow_read( fd, data, length );
    } else {
        return read( fd, data, length );
    }
}};
shadow_action
{{
    return read( fd, data, length );
}};

ssize_t write( int fd,
               in opaque "length" const void *data,
               size_t length )
agent_action
{{
    if( fd<3 ) {
        return bypass_shadow_write( fd, data, length );
    } else {
        return write( fd, data, length );
    }
}}
shadow_action
{{
    return write( fd, data, length );
}};
```

Figure 4. Agent for Remote I/O

4.1.1. Remote I/O

Our first example is a simple remote I/O system. This agent traps operations performed on the standard input, output, and error streams, and sends them to the shadow on the home machine for execution. Other operations are passed through to the standard library without modification. The net effect is to leave file I/O untouched, while sending logging and error output back to the home environment for monitoring by the user.

Figure 4 shows the Bypass source code for this system. Two declarations are provided: one for

```
agent_prologue
{{
    static int bytes_read=0;
    static int bytes_written=0;
}};

ssize_t read( int fd,
              out opaque "length" void *data,
              size_t length )
agent_action
{{
    int result;
    result = read(fd,data,length);
    if(result>0) bytes_read+=result;
    return result;
}};

ssize_t write( int fd,
               in opaque "length" const void *data,
               size_t length )
agent_action
{{
    int result;
    result = write(fd,data,length);
    if(result>0) bytes_written+=result;
    return result;
}};

void exit( int status )
agent_action
{{
    printf("%d bytes read, %d written\n",
           bytes_read, bytes_written );
    exit(status);
}};
```

Figure 5. Agent for Measuring I/O

```
agent_prologue
{{
    extern "C" {
        @include "globus_common.h"
        @include "globus_gass_file.h"
    }
}};

int open( in string const char *path, int flags, int mode )
agent_action
{{
    globus_module_activate( GLOBUS_GASS_FILE_MODULE );
    return globus_gass_open( path, flags, mode );
}};

int close( int fd )
agent_action
{{
    return globus_gass_close( fd );
}};
```

Figure 6. Agent for Attaching GASS

`read` and one for `write`. For each, parameters are declared in a manner similar to C. The second argument for each points to data to be transferred and so is labelled with `in` or `out`, corresponding to the direction of data flow, followed by `opaque "length"` indicating the kind and amount of data to be transferred. The quoted value may be any valid C expression that can be evaluated at runtime to yield the number of bytes to transfer. In this case, the exact number is given by the third argument, named `length`.

Two procedure bodies follow, one for the agent, and one for the shadow. The bodies may contain any arbitrary C code, including references to the original procedures, or to the RPC routines, which are prefixed with `bypass_shadow`. In this case, the body considers the value of the file descriptor. If it is less than three, it refers to a standard stream, so the RPC is performed, otherwise the original procedure is invoked.

4.1.2. Measuring I/O

Not all agents are necessarily paired with a shadow. A number of useful tasks can be performed without invoking any RPCs. Bypass can also create such standalone agents using the same language.

Accurately measuring program behavior is critical to designing appropriate distributed systems. A standalone agent for measuring the I/O performed by an application is shown in figure 5. It traps the `read` and `write` procedures, invokes the original versions, and then records how many bytes were transferred before returning control to the application. When the application calls `exit`, the agent prints a short message summarizing the I/O activity. Procedures not defined by the agent, such as `open` and `close`, are passed through to the standard library below.

4.1.3. Attaching a Filesystem

Deploying a new distributed filesystem is a difficult matter. Most filesystems are implemented as elements of an operating system kernel so as to be transparently available to all processes. However, kernel code is generally not portable and a kernel makes for a difficult development and debugging environment. Furthermore, users of the system must have administrator privileges on each machine they wish to use the new filesystem on.

An alternative is to implement a filesystem at the user level. This is attractive, especially in distributed systems, because user-level code is much easier to port to multiple systems and can be applied by nearly any user, regardless of privileges. However, attaching a user level filesystem to an application may require re-writing or re-compiling the application, which is not an acceptable solution for reasons given above.

This problem can be solved by building an agent which translates standard I/O operations into their equivalents in a user-level filesystem. In this manner, any application can take advantage of the new system, but no special privileges are required to do so.

An example of a user-level file system is Global Access to Secondary Storage (GASS) [4]. This is a library with two entry points, `globus_gass_open` and `globus_gass_close`, which have the same signature as POSIX `open` and `close`. If an ordinary filename is opened using this library, the two procedures will simply call `open` and `close`. However, if the filename specifies a distributed resource, the library will make a local copy of the file and return a file descriptor pointing to the local copy. A number of features, such as caching and strong authentication, are pleasant side effects.

A very simple agent can attach GASS to an arbitrary application. The agent must simply trap `open` and `close` and then invoke either `globus_gass_open` or `globus_gass_close` with the same arguments. This agent is depicted in figure 6. Notice that both `open` and `close` make (indirect) use of many system calls in the library below in order to implement their complex behavior.

4.2. Semantics

A number of agents may be applied at once to a single application. The application, agents, and standard libraries form a stack of software through which calls percolate. We call each of these software elements a *layer*. In most programming systems, it is an error to define a procedure multiple times; with Bypass, each layer may have its own definition of a symbol. All these definitons can be a little confusing, so we must establish some rules to clarify the binding of calls to definitions.

1. A process keeps track of its *active layer* in a global variable. A process begins execution with the top layer active.
2. A call to a trapped procedure name selects the definition in the layer immediately below the active layer. If none is present, the next layer below is searched, and so on.
3. After selecting but before invoking a trapped procedure, the active layer is lowered to that of the selected definition. Before returning, the active layer is restored to its previous value.
4. A call to a non-trapped procedure does not consult or affect the active layer. Such calls are bound according to the normal linking policy of the operating system.

It should be emphasized that these rules bind names to procedures based on the run-time value of the active layer variable. The location of a procedure call is irrelevant to the binding.

To illustrate this, consider the use of `printf` in figure 5. `printf` formats some text and then sends it to the standard output stream using `write`. `printf` is not trapped by any of the agents, so calling it does not consult or affect the active layer. [1] Which definition of `write` does `printf` call? It depends on the layer from which `printf` itself was called. If `printf` is invoked from the application

[1] Technically, it is a member of the standard library, but because it is not trapped by an agent, it is not logically a member of that layer.

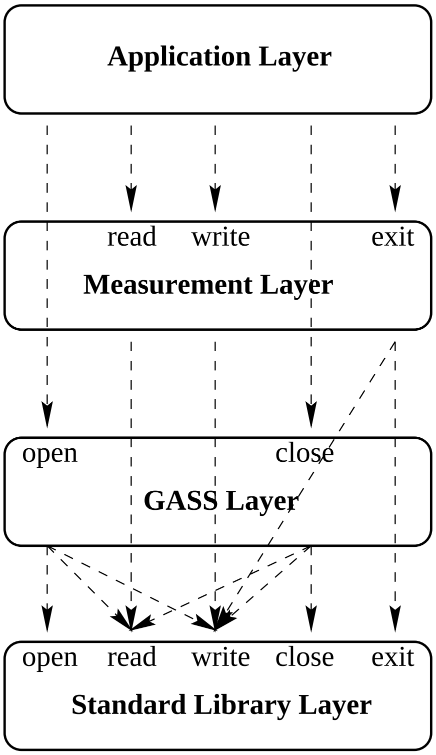

Figure 7.

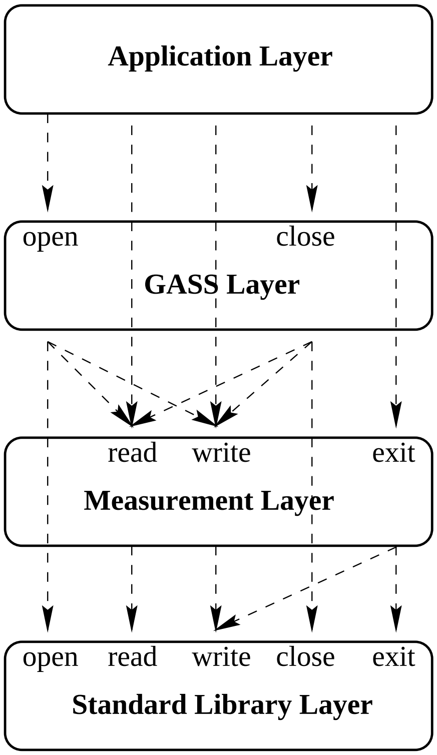

Figure 8.

layer, then the `write` will be bound to the topmost agent layer. If `printf` is invoked from the topmost agent layer, then the `write` will be bound to the layer below.

These rules also describe how multiple agents can be applied to the same program. The layering of multiple agents is also called *composition*. In general, composition is not commutative: the order in which agents are layered affects their semantics. Other work [3] has suggested, without defining the term, that agents may commute if they are *disjoint*. We wish to clarify this with an example.

The measurement and GASS layers described above would appear at first glance to be disjoint. One only traps `read` and `write`, while the other only traps `open` and `close`. However, they may be layered in one of two ways, yielding distinctly different systems. The two possibilities are shown in figures 7 and 8.

In figure 7, the measurement layer is placed above the GASS layer. If the application invokes a `read` or `write`, it will be measured and then passed down to the standard library. If an `open` is invoked, it passes through the measurement layer and is trapped by the GASS layer. The implementation of this call involves all sorts of procedure calls – including `read`s and `write`s – which are passed through to the standard library without measurement. This ordering causes the measurement layer to record the actions attempted by the application but not any actions taken by layers below it.

The opposite ordering is shown in figure 8. In this case, both `read`s attempted directly by the application and indirectly through a call to `open` are measured. This ordering causes the measurement layer to record the operations actually performed by all of the layers above.

Both arrangements are useful. In a stack of multiple agents, it would be worthwhile to install one measurement agent as the topmost, and one as the bottommost. The topmost measures the activity of the application independent of the transforming layers below. The bottommost measures the actual resources consumed by all of the software above.

The GASS and measurement layers would appear to be disjoint because they trap a disjoint set of procedures. They are not disjoint because the GASS layer invokes procedures trapped by the measurement layer, namely `read` and `write`.

The original statement still stands: disjoint agents may commute. However, we propose that two agents are only disjoint if the set of procedures they trap *and* invoke are disjoint. Because even simple agents invoke code from other libraries, it can be difficult to determine whether any two nontrivial agents are truly disjoint.

Our rules for combining agents are quite strict. They ensure that a given layer can only be invoked by layers above and is only capable of invoking layers below. With strict layering in effect, a measurement layer can successfully determine exactly what operations are performed by those layers above. The character of the measurement may be changed by re-ordering the layers.

Other rules for combining agents are certainly possible. For example, Mediating Connectors [3] allows an agent to invoke either an *inner* or an *outer* call. In Bypass parlance, an inner call corresponds to invoking the next layer below, while an outer call corresponds to re-invoking the procedure at the topmost layer.

It is not clear if our rules are any more or less "correct" than others. We note that permitting outer calls prevents the construction of sensible measurement layers. If an lower layer may invoke a layer above, a measurement layer in between will produce results that are quite difficult to interpret. On the other hand, we can envision situations where re-invoking a layer above might be useful. For example, a low-level layer such as a device driver might invoke a high-level procedure to produce a dialog box that communicates an error.

4.3. Implementation

Bypass is a pre-processor, much like the tools lex [13] and yacc [11]. It reads a source file written in the language described above, combines it with a *knowledge file* describing how to trap certain calls, and then produces C++ source code for the corresponding agent and, if necessary, shadow.

The knowledge file encapsulates many of the difficulties we have outlined above. When the programmer requests that a particular call be trapped, the knowledge file details the multiple entry points to trap and the necessary type conversions needed to accomplish the programmer's goal. In addition, a library is provided which gives canonical external forms for POSIX constructs, ranging from the `stat` structure to the values for flags such as `O_CREAT`. This allows RPCs to be performed between systems that implement the standard with different values. We have concentrated on making the knowledge file apply to the POSIX interface, but we know of no technical obstacle to using Bypass on another interface such as ANSI C functions or the X Windows library. By varying the knowledge file, we have successfully ported Bypass to several versions of Linux, Solaris, IRIX, and Digital Unix.

Each agent is compiled into a shared object which can easily be inserted into an application at run time. On most UNIX-like systems, this is accomplished by setting an environment variable which is consulted by the dynamic linker. Any number of agents may be specified in the order, top to bottom, that they should be layered. The application's standard libraries are implicitly added to the end of the list. For example, to load our three example agents on a Linux system, one may execute:

```
LD_PRELOAD="measure.so gass.so remoteio.so"
```

We have selected this mechanism for several reasons. It allows the trapping of procedure calls that are not necessarily system calls. It can be used by any user without special privileges or kernel changes. It is a *de facto* standard across most UNIX-like systems, discounting some syntactic variations. Finally, it may be used on a wide variety of existing dynamically linked programs.

This mechanism is well-suited towards use in a distributed system. A user sending programs around the world via a metacomputing framework cannot reasonably expect all participating machines to modify their kernels or grant the user an administrator account. A user with normal privileges can apply this mechanism without any help from the owner of a machine.

This mechanism is not suitable for all purposes. It cannot be used as a security mechanism, because the application may simply choose to use other procedures or to invoke system calls directly. It also cannot operate on statically-linked programs. Such situations are better handled by binary rewriting or a kernel-level mechanism.

Each agent defines one public entry point for each procedure it wants to trap. This entry point is called a *switch*: it decides what layer is active according to the rules given above, looks up the correct procedure, and transfers control there. The `agent_action` for that entry point is placed in a separate procedure. Each agent also maintains the complete list of agents and a pointer to the active layer. The agents rely on the dynamic linker to provide routines for searching and invoking procedures in shared objects.

When the linker loads a list of agents into an application, it binds procedure calls in the application to the topmost switches in the stack of agents. This is done according to the linker's normal rules: it binds name references to the first available definition. All others (from the linker's point of view) are ignored. Each agent defines a switch for each of its entry points and code to keep track of active layer. However, only the topmost agent actually manages the active layer, any only the topmost switch for each trapped function are actually used. This allows the agents to be arranged in any order at run time, but comes at the cost of some code duplication.

4.4. Performance

We have constructed a synthetic testing program to measure the overhead incurred by Bypass, independent of its application. The test was run in six configurations. The results for the first five are given in figure 9. The testing program simply invokes each system call a large number of times in a tight loop. The "open/close" test opens and closes the same file without any intervening operations. "stat" returns metadata about a file. "getpid" gets the current process identifier. Finally, reads and writes of one byte and eight kilobytes are performed to a file. In all cases, the files were in `/tmp` and cached in memory so as to avoid any perturbances due to physical storage.

In the first configuration, the testing program was run with no interference from Bypass. In the second, a Bypass agent trapped each system call and re-invoked it without modification at the foreign machine. For example, the specification for `close` was:

```
int close( int fd )
agent_action {{
    return close(fd);
}};
```

In the normal operation of Bypass, the first reference to a trapped function in an agent requires a symbol lookup using the system's dynamic library manipulation tools. The function pointer is then cached for later reference. In the second configuration, caching was enabled for normal operation. The third configuration ran the same tests, but with caching disabled, thus measuring the actual cost of an initial lookup. In the fourth configuration, identical agents were layered one by one up to a total of ten. A linear fit was performed, yielding a slope which measures the average cost of adding an agent past the first. In the fifth configuration, a Bypass agent trapped each system call and sent it via RPC to a shadow on the same machine to be executed. For example, the specification for `close` was:

```
int close( int fd )
agent_action {{
    return bypass_shadow_close(fd);
}}
shadow_action {{
    return close(fd);
```

System Call	Unmodified Program	Execute At Agent (Caching)	Execute At Agent (No Caching)	Cost per Addl. Agent (Caching)	Execute At Shadow
open/close	27.7	36.1	59.7	2.65	914
stat	18.5	24.2	36.6	2.50	621
getpid	2.4	4.5	11.3	1.05	406
read 1 byte	12.0	14.7	27.4	1.27	445
write 1 byte	13.6	18.0	28.4	1.23	463
read 8 KB	53.6	54.1	81.5	1.19	988
write 8 KB	64.6	67.1	81.0	1.26	1019

All times are given in microseconds.
Variance is less than five percent for all entries.

Figure 9. System Call Overhead

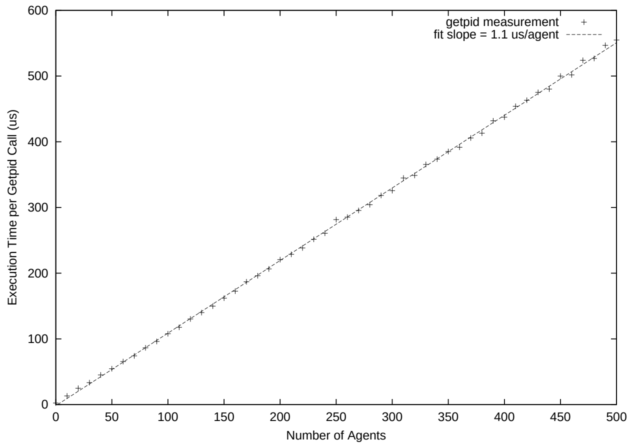

Figure 10. Scalability of Layering Agents

```
}};
```

In the final configuration, we applied as many agents as possible to test to limits of layering. By creating an agent with a one-letter name, we were able to demonstrate linear behavior trapping `getpid` with up to 500 agents, at which point we reached the maximum length of the preloading environment variable. The results are shown in figure 10.

In each configuration, the wall clock time was measured for 100,000 iterations of each system call. This process was repeated 10 times, giving a mean and variance. Variance for each measurement was less than five percent of the mean. The values reported in figure 9 are the means divided by the number of iterations, yielding the time necessary for a system call in the given configuration. The testing machine was a 200 MHz Pentium Pro workstation with 128 MB of memory and running Solaris 2.6.

The results meet our goals. Trapping a system call at the agent is quite fast: the first call to a trapped procedure takes between 9 to 28 μs, while later invocations incur only 2 to 5 μs. If multiple agents are layered, calls to deeper layers cost from 1 to 3 μs. These time variations may be attributed to the varying amount of data to be copied as parameters, variations due to the name lookups performed by the dynamic linker, and the miscellaneous conversions and extra trapping points provided by the knowledge file. Sending the system call via RPC to be executed at the shadow is an order of magnitude slower, and depends heavily on the network and application. The programmer using Bypass can conscientiously use the expensive remote procedure call when necessary, but does not pay a significant cost for trapping a system call and deciding to execute it locally.

Our results are a few microseconds slower than those given previously [2]. The earlier work only allowed one agent to be applied at once, so the slowdown can be attributed to the newer mechanism which manages several agents at once. The `stat` test is actually faster, due to an inefficiency discovered and removed from the knowledge file.

5. Related Work

As we have noted, agents may be implemented in a number of ways. It is interesting to observe that interposition agents on UNIX-like systems have generally trapped calls at the kernel interface, while those on Windows systems have generally trapped at the standard library interface.

The term *interposition agent* was coined by Michael Jones [12] to describe a technique for placing software between a program and the operating system kernel. This system relied on the Mach facility to reroute calls to user-level code in the same process. Jones provides an object-oriented interface to the structures exported by the kernel, such as pathnames and file descriptors, while most other systems work in terms of the calls to be trapped.

SLIC [7] is similar in functionality to Bypass, but uses an interposition mechanism inside the kernel. The mechanism cannot be diverted, and so is appropriate for the problem domains suggested in that work: security patches, sandboxing, and encryption. It is not suitable for our problem domain – distributed systems – for three reasons. First, it requires superuser privileges to install the (admittedly small) kernel hooks to support extension. Second, it does not provide a platform-independent means of specifying the operation of an agent. Finally, it does not permit the redirection of procedure calls that are not system calls. We have noted above that correct split execution requires trapping plain procedure calls.

The UFO system [1] relies on a kernel facility to monitor the system calls of one user level process from another. This method shares the same advantages SLIC has over Bypass and additionally can be used by any user without special privileges. However, the mechanism incurs a high overhead (trapped calls are 4-7 times slower) and can only be applied at the kernel interface.

Bypass shares its title metaphor with Detours [9], a system for intercepting calls to Windows library procedures. Detours uses binary rewriting to intercept the flow of control, and so can be applied to any sort of program at all. Bypass relies on the system's dynamic linker, and thus can only be used to intercept public, dynamically linked procedures. The main contribution of Detours is to make the un-instrumented target function available through a special mechanism called a *trampoline*. This is roughly comparable to the *switch* in Bypass, although the Detours mechanism is called by a name distinct from the original entry point. By preserving the original procedure name, Bypass allows the use of unmodified utility routines (such as `printf`) within an agent.

A similar package, Mediating Connectors [3], also traps Windows library procedures, but permits the composition of multiple agents. The rules used by this package are compared with those of Bypass above in section 4.2.

Remote procedure calls are a standard technique [5,17,15]. Our facility is similar to other implementations, but is driven by the need for drop-in software which works without modifying the target application. To this end, our specification syntax relies on annotating existing interfaces instead of creating new protocols from scratch. Likewise, the RPC client implicitly configures and connects at the first use of an RPC routine, using user-supplied environment variables to select the shadow address. Our external data representation is also quite similar to existing standards [16], but goes beyond specifying integer size and endianness. To provide cross-platform operation, we must provide consistent value semantics by transforming symbolic constants into canonical values.

6. Conclusion and Future Work

Bypass is a general-purpose tool and language for building split execution systems. We build upon previous work by providing an implementation independent language, semantics for combining agents, and an implementation suitable for distributed computing. There is more work to be done in each of these areas.

The Bypass language allows agents to be specified without regard to the underlying implementation. Both trapped and untrapped procedure calls may be invoked by their original names, thus allowing the use of standard subroutines and libraries within agents. It would be useful to also express interposition on signal propagation in the same language. This presents several problems:

1. Signals travel *up* through interposition layers.
2. The semantics for combining signal-trapping agents are yet undefined.
3. Trapping signals may require trapping all the various standard procedures that manage the signal disposition of a process.

We have proposed very strict rules for combining agents. These prevent complex interactions, allowing the construction of agents that can make useful measurements of calls between layers. However, less restrictive rules may be needed to implement certain applications.

Bypass uses library preloading to insert agents. This is suitable for the purpose of equipping applications to run correctly in a distributed environment. Other mechanisms, such as in-kernel facilities, are more suitable for implementing a security policy. We envision a complete distributed system that includes both: the owner of a machine might provide a heavy-duty kernel-level sandbox to protect the machine from a foreign program, while the owner of a migrating program might provide a light-weight user level agent to assist the execution of the program. In this case, the two varieties of agents may need to negotiate in order to determine mutually acceptable operations.

Software, manuals, and further information about Bypass may be downloaded from
`http://www.cs.wisc.edu/condor/bypass`.

References

[1] Albert Alexandrov, Maximilian Ibel, Klaus Schauser, and Chris Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[2] Douglas Thain and Miron Livny. Bypass: A tool for building split execution systems. In *Ninth IEEE Symposium on High Performance Distributed Computing*, pages 79–85, August 2000.

[3] Robert Balzer and Neil Goldman. Mediating connectors. In *19th IEEE International Conference on Distributed Computing Systems*, June 1999.

[4] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. *6th Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Februrary 1984.

[6] I. Foster and C. Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[7] Douglas P. Ghormley, Devid Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference*, June 1998.

[8] A.S. Grimshaw, W.A. Wulf, et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[9] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. Technical Report MSR-TR-98-33, Microsoft Research, February 1999.

[10] IEEE/ANSI. Portable operating system interface (POSIX): Part 1, system application program interface (API): C language, 1990.

[11] S.C. Johnson. YACC – Yet another compiler-compiler. Comp. Sci. Tech Rep. 32, Bell Labs, Murray Hill, New Jersey, July 1975.

[12] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM symposium on operating systems principles*, pages 80–93, 1993.

[13] M.E. Lesk and E. Schmidt. LEX – a lexical analyzer generator. Comp. Sci. Tech. Rep. 39, Bell Labs, Murray Hill, New Jersey, 1975.

[14] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[15] R. Srinivasan. RFC-1831: RPC: Remote procedure call protocol specification version 2. *Network Working Group Requests for Comments*, August 1995.

[16] R. Srinivasan. RFC-1832: XDR: External data repre-

sentation standard. *Network Working Group Requests for Comments*, August 1995.

[17] Sun Microsystems. `rpcgen` *Programming Guide*. Sun Microsystems Inc., Mountain View CA, 1987.

7. Biography

Miron Livny received a B.Sc. degree in physics and mathematics in 1975 from the Hebrew University and M.Sc. and Ph.D. degrees in computer science from the Weizmann Institute of Science in 1978 and 1984, respectively. Since 1983 he has been on the Computer Sciences Department faculty at the University of Wisconsin-Madison, where he is currently a Professor of Computer Sciences. He has been leading the Condor project since 1986.

Douglas Thain received a B.S. degree in physics in 1997 from the University of Minnesota-Twin Cities and a M.S. degree in computer sciences in 1999 at the University of Wisconsin-Madison. He is currently a Ph.D. student with the Condor project at UW-Madison and specializes in distributed I/O systems.