Bypass: A Tool for Building Split Execution Systems

Douglas Thain and Miron Livny

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

Abstract

Split execution is a common model for providing a friendly environment on a foreign machine. In this model, a remotely executing process sends some or all of its system calls back to a home environment for execution. Unfortunately, hand-coding split execution systems for experimentation and research is difficult and error-prone. We have built a tool, Bypass, for quickly producing portable and correct split execution systems for unmodified legacy applications. We demonstrate Bypass by using it to transparently connect a POSIX application to a simple data staging system based on the Globus toolkit.

1. Introduction

The split execution model allows a process running on a foreign machine to behave as if it were running on its home machine. Split execution generally involves three software components: an application, an agent, and a shadow. Figure 1 shows these components.

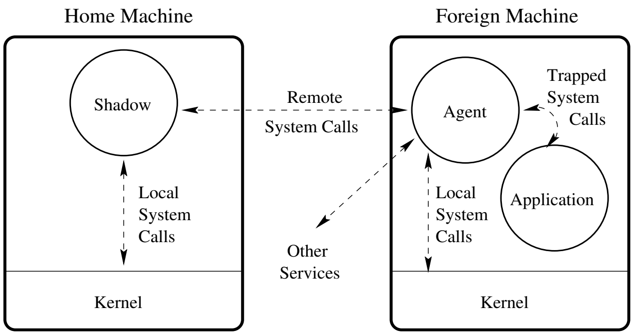

Figure 1. Overview of Split Execution

A *foreign machine* is a machine that has given permission for a particular application to use its CPU and memory, but may or may not have other resources needed by the application. A *home machine* is a machine on which the application could run correctly. A *resource* is any item that is accessed via system calls – this most often refers to files, but also includes items such as network connections, timers, and system databases. The *application* is a normal program running unmodified on the foreign machine. The application is supervised by the *agent*, which traps its system calls and routes them to various services, including the shadow, the foreign kernel, or perhaps other third party services. The *shadow* serves to execute the application's system calls on the home machine and returns the results to the foreign machine, enabling the application to run as if it were on the home machine.

This model has been used in a wide variety of kernel-level [4, 13, 2] and user-level [12, 11, 7] distributed systems. However, split execution remains an open research topic because many variations on the basic idea are possible. For example, data may be lazily or aggressively cached between the agent and the shadow. Policy decisions regarding system 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.

In this paper, we discuss the difficulties of implementing such systems and our vision of an ideal split execution framework. We describe Bypass, a tool we have created for building a wide variety of split execution systems. Bypass does not implement any particular system, but allows flexible construction of shadows and agents without requiring the programmer to re-implement any of the difficult system dependencies we have discovered. We conclude with an example of using Bypass to build a data staging system based on the Globus toolkit.

2. Difficulties

Speaking from the experience of developing the Condor system, we assert that hand-coding a portable split execution system 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.* The `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()`.

2. *Varied implementations.* `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.

3. *Binary incompatibilities.* 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

3. A Framework for Split Execution

We envision a system where the programmer writes a *specification* which describes, in plain C, mechanisms for handling system calls at the agent and shadow. The system should provide the *knowledge* which describes the peculiarities of trapping and forwarding system calls on various operating systems. A code generator should examine the user's specification and create complete source code for a conforming shadow and agent.

Our goals for this framework 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 for split execution 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 for split execution should 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 have shown that trapping system calls for split execution involves knowledge of unpleasant implementation details. The programmer of the shadow and agent is not interested in creating or dealing with this knowledge for every new program. Tools for split execution should combine the programmer's expressed intent implicitly with the details needed to implement the system.

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 system call trapping mechanism itself should not cause a significant slowdown. System calls directed to the foreign machine by the agent should run at nearly native speed.

4. Bypass

Bypass is a tool for creating split execution systems. Bypass reads two input files, a specification file and a knowledge 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.

4.1. Writing Bypass Code

The specification file, provided by the programmer, names the system calls to be trapped by the agent, describes the data transfer needed for the parameters, and gives the code that is to be used in place of each system call at both the agent and the shadow. Figure 3 shows a specification that might be used for `open()`. This example gives a simple policy for handling files opened by an application: If the file begins with `/tmp`, open the file at the foreign machine, otherwise open it at the home machine and emit a brief message there.

```
int open( in "_POSIX_PATH_MAX" const char *path,
          int flags,
          int mode )

    agent_action
    {{
        if(!strncmp(path,"/tmp",4)) {
            return open(path,flags,mode);
        } else {
            return bypass_shadow_open
                       (path,flags,mode);
        }
    }}
    shadow_action
    {{
        printf("program opened %s\n",path);
        return open(path,flags,mode);
    }}
    ;
```

Figure 3. Example Specification File

Each entry in the specification looks like standard C, with a few notable exceptions.

The function header gives the name of the system call (`open`) and the names and types of its parameters. The `path` parameter is annotated with `in "_POSIX_PATH_MAX"`, indicating the number of bytes to be transferred to the shadow when a remote procedure call (RPC) is necessary. All pointer arguments must be annotated like this to clarify the ambiguities inherent in a C interface.

There are *two* function bodies in each specification.

The `agent_action` body gives the code that is to be executed by the agent every time the application attempts a system call. This function body may include calls to the original procedure that the agent replaced, or remote procedure calls to the shadow. Each of these operations are demonstrated by calls to `open()` and `bypass_shadow_open()` in the example. If the `agent_action` is omitted, it is assumed to be "invoke the `shadow_action` by RPC".

The `shadow_action` body gives the code that is executed at the shadow in the event of a remote procedure call. If the `shadow_action` is omitted, it is assumed to be "invoke the real system call here."

4.2. Using Bypass Code

The code generator creates source code for an agent and a shadow. The agent is compiled into a shared library, while the shadow is compiled into a standalone executable. To split and run an application, three steps are necessary. 1 - The shadow is invoked in the home environment, where it will listen on a well-known port for the agent to connect. 2 - The foreign environment is prepared by placing the network address of the shadow into an environment variable and instructing the dynamic linker to "pre-load" the agent library. On most UNIX-like systems, this is accomplished by setting the environment variable `LD_PRELOAD` or `_RLD_LIST`. 3 - The application is invoked as normal.

4.3. Implementation of Bypass

4.3.1. Trapping System Calls

Interoperability requires that system calls be trapped at a well-known interface, such as those defined by POSIX. We have noted above that standard interfaces (such as `socket`) are not necessarily system calls, so our trapping mechanism must be capable of intercepting simple procedure calls. A Bypass agent "traps" calls merely by virtue of being linked with the application before any other library. This can be done by explicitly linking the program with the library at build time, or, as noted above, by using the system linker to force the library into an existing executable. At the first invocation of a trapped system call, the agent will examine the environment for the shadow's network address, and make the necessary connection. System calls will then be executed according to the specification.

4.3.2. Knowledge

The knowledge file contains a laundry list of exceptions and special cases known about particular operating systems and libraries. Figure 4 shows simple examples of some of these entries. The default entry, denoted by `*`, indicates that for any system call not otherwise specified, additional entry points should be generated for the same name with `_` and `__` prepended. The `also` clause in the entry for `open()` indicates that whenever the programmer wants to replace `open()`, the code enclosed in {{ }} should be included. In this case, the additional code is for trapping calls to `creat()`, which is simply a restricted interface to the same functionality as `open()`. A variety of other structures for listing exceptions are detailed in the Bypass manual. [16]

```
options "*"
    entry "_*", "__*"
    ;

options "open"
    also
    {{
        int creat( const char *path, mode_t mode ) {
            return open( path,
                         O_WRONLY|O_CREAT|O_TRUNC,
                         mode );
        }
    }}
    ;
```

Figure 4. Example Knowledge File

We certainly do not claim that our collection of "knowledge" is complete! Our experience is that every new release of an operating system contains surprises in the standard library that must be understood and folded into the knowledge file. However, we expect that the typical programmer working on a supported platform will not consult or edit the knowledge file in day to day operations.

4.3.3. Emitted Code

For each system call specified by the programmer, Bypass emits a number of procedures. The relation between all the procedures is shown in figure 5.

The *entry points* are a number of functions which encompass all of the myriad ways to invoke a given system call. These entry points are never specified by the user, but are provided by the knowledge file. In the example shown, these entry points include `open()`, `_open()`, `__open()`, and `creat()`.

Each of the entry points simply invokes the *switch* with the arguments to the system call. The switch examines the current *system call mode*, which is a global variable with one of two values: `LOCAL` or `REMOTE`. When the system call mode is `LOCAL`, the switch invokes the corresponding original system call and returns. When the system call mode is `REMOTE`, the switch changes the mode to `LOCAL` and invokes the agent action. Before returning, the switch changes the mode back to `REMOTE`. By default, the system runs in `REMOTE` mode.

The *agent action* is the actual code the programmer intends to run on the foreign machine in place of the given system call. The agent action can be composed of any arbitrary code, but, because it is run in `LOCAL` mode, any references to system calls (such as `open()`) will be routed by the switch to the original implementation of the system call. This allows the agent action to make use of existing subroutines that expect `open()` to be defined in the usual way.

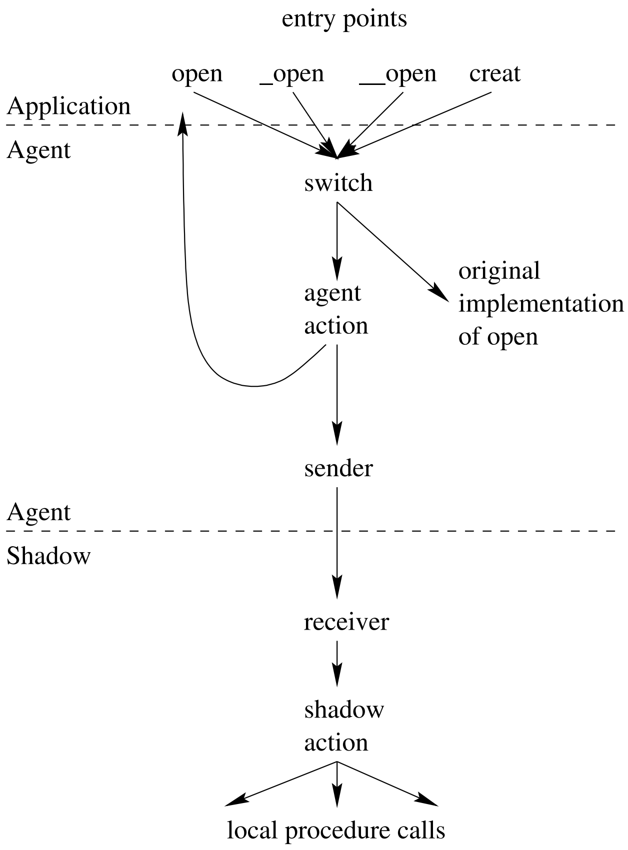

Figure 5. Code Structure for One System Call

The agent action may optionally perform remote procedure calls to the shadow. These procedure calls are implemented in three parts. The *sender* transforms the procedure arguments into an external representation and transmits them to the shadow. In the shadow, a corresponding *receiver* transforms the external representation into a local form and then invokes the *shadow action* given by the programmer. The shadow action may perform any arbitrary computation and return a result back to the agent.

4.3.4. External Representation

In order to achieve our goal of interoperation between dissimilar machines, we must carefully choose an external representation which is understandable by all participants.

As noted above, POSIX interfaces are often defined at the source level, resulting in binary incompatibilities. Because Bypass is explicitly designed for trapping at the POSIX interface, it contains knowledge of POSIX constructs and converts them into a portable external representation. The three aspects of this representation are:

- *Uniform integer format.* All integers are represented as 64-bit signed integers in network byte order.
- *Uniform symbol values.* All integers known to contain symbolic constants, such as the bit fields accepted by `open()`, are converted into canonical values.
- *Uniform structure encoding.* System interface structures are encoded in canonical orders, regardless of the ordering of the structure on each platform.

In each of these three aspects, a receiver must decode the external representation into the best data type available locally. If this cannot be done, then the system call will fail with an appropriate error. For example, one system may represent file offsets with 64 bits while another uses 32. These systems will be able to interact so long as an offset never actually reaches a value that cannot be represented in 32 bits.

5. Performance

We constructed a synthetic testing program to measure the overhead incurred by Bypass. The results are given in figure 6. 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.

System Call	Unmodified Program	Execute At Agent	Execute At Shadow
open/close	28.2	32.7	914
stat	48.0	52.1	621
getpid	2.4	3.1	406
read 1 byte	12.2	13.9	445
write 1 byte	13.6	16.1	463
read 8 KB	54.5	57.7	988
write 8 KB	66.5	69.2	1019

All times are given in microseconds.

Figure 6. System Call Overhead

The test was run in three configurations. 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 third, 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 )
        shadow_action {{ return close(fd); }};
```

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 standard deviation. Standard deviations for each system call were less than five percent of the mean. The value reported in figure 6 is the mean 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 – 3 to 4 μs – because the trapping mechanism is merely a function call. Sending the system call via RPC to be executed at the shadow is an order of magnitude slower, and could be much worse on a wide-area network. 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.

6. Related Work

Bypass shares its title metaphor with Detours [9], a system for intercepting calls to 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, which makes the target function available through its original entry point.

An agent created by Bypass is an example of an *interposition agent*. This term was coined by Michael Jones [10] to describe a technique for placing software between a program and the operating system kernel. Although we have built Bypass specifically to understand structures in the POSIX software layer, it can be used to intercept calls between any procedures, not just those adjacent to the kernel. Jones provides an object-oriented interface to many of the structures exported by the kernel, while Bypass simply exposes a procedural interface to individual calls.

The UFO system [3] uses an interposition agent called the Catcher to attach user-level file systems to arbitrary processes. The Catcher relies on a kernel facility to monitor the system calls performed by another process. This method has several advantages over Bypass: it can be used on any program at all, and it can be used as a security mechanism because it cannot be worked around. On the other hand, the mechanism incurs a high overhead (trapped calls are 4-7 times slower) and can only be applied at the kernel interface. As noted above, split execution sometimes requires trapping of procedures that are not kernel calls.

RPC is well described in the literature. [6, 1, 14] 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 RPC client implicitly configures and connects at the first use of an RPC routine. The address of the server is provided by the user externally through the use of environment variables. Our external data representation is also quite similar to existing standards [15], 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.

7. Example Application: Data Staging

We have used Bypass to create a simple data staging system for legacy applications. An unmodified application runs on a computation node which is assumed to have limited disk space, and thus is not capable of staging large data files. The agent traps the application's I/O operations and sends them to a shadow running on a high capacity server. The shadow is responsible for staging the application's data files, using the GASS library [5] from the Globus toolkit [8]. An overview of this application is given in figure 7.

This example is trivial to build using Bypass. The *entire* specification file for this system is given in figure 8. The standard POSIX operations `open()`, `close()`, `read()`, `write()`, and `lseek()` are sent by RPC to the shadow. There, the `open()` and `close()` operations are mapped to their analogues in the GASS library, while the others are executed unmodified.

Certainly, this example system is not complete. One can imagine improving it by adding data buffering or any number of other features. We feel that Bypass is a flexible tool for implementing such features without getting bogged down in the implementation details of system call trapping and forwarding.

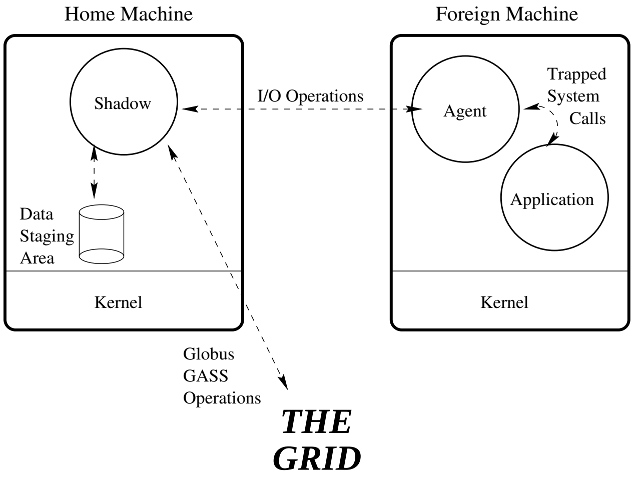

Figure 7. Split Execution for Data Staging

8. Conclusion

In theory, split execution is a convenient model for remote execution. In practice, it is very difficult to cover the necessary details to create an interoperable and portable system. Bypass is a tool which hides these practical matters and allows the programmer to concentrate on the higher-level problems of split execution.

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

```
shadow_prologue
{{
    @include "globus_common.h"
    @include "globus_gass_file.h"
}};

int open( in "_POSIX_PATH_MAX" const char *path,
          int flags,
          int mode )

    shadow_action
    {{
        globus_module_activate
            ( GLOBUS_GASS_FILE_MODULE );
        return globus_gass_open
            ( name, flags, mode );
    }};

int close( int fd )

    shadow_action
    {{
        return globus_gass_close( fd );
    }};

int read(    int fd,
             out "length" void *data,
             size_t length );

int write(   int fd,
             in "length" const void *data,
             size_t length );

off_t lseek( int fd,
             off_t where,
             int whence );
```

Figure 8. Specification for Data Staging

References

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

[2] R. Agrawal and A. K. Ezzat. Location independent remote execution in nest. *IEEE Transactions on Software Engineering*, 13(8):905–912, August 1987.

[3] A. Alexandrov, M. Ibel, K. Schauser, and C. 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.

[4] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[5] 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.

[6] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Februrary 1984.

[7] J. Cruz and K. Park. DUNES: A performance-oriented system support environment for dependency maintenance in workstation networks. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 309–318, August 1999.

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

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

[10] M. 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.

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

[12] M. J. Litzkow. Remote unix - turning idle workstations into cycle servers. In *USENIX Conference Proceedings*, pages 381–384, Summer 1987.

[13] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.

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

[15] R. Srinivasan. RFC-1832: XDR: External data representation standard. *Network Working Group Requests for Comments*, August 1995.

[16] D. Thain. Bypass manual. Available from `http://www.cs.wisc.edu/condor/bypass`, 2000.