A Hardware-Independent Parallel Operating System Interface

by Sean Halle

January 15, 2007

Abstract

This paper describes an interface to OS services that is free of hardware implications, allowing the same distribution bundle to be installed on hardware ranging from a laptop to a NUMA supercomputer. An OS defines an application's "view of the world", so the abstractions used to model the world must be hardware-implication free. Such an interface is difficult to define because hardware implications arise in so many places.

The proposed OS interface overcomes the difficulties with: 1) its four abstractions used to model the universe for the app, 2) the separating of pure name from attached information, 3) adaptors, 4) translation-at-install, and 5) an intermediate format that is a "dynamic" circuit, with spontaneously created short-lived processors that do the work.

The four abstractions that model the universe are data, processor, specification, and name-space. Everything in the OS-provided view of the world is one of these four things.

Names, such as paths in Unix, commonly contain both meta-data used to search for desired data, and also location information used by the name-space mechanism to access the data. Separating this information, by using "pure" names, hides name-space mechanism information from the application.

Adaptors and translate-at-install enforce a data-format boundary between external formats and internal formats.

The dynamic circuit computation model has atomic operations for which processors come into existence, perform the operation on one piece of data, then go out of existence. The ordering and overlap of the existence of the processors is undefined. This, in combination with other features, allows code to be re-partitioned automatically during translate-at-install. The translation results in units of code whose complexity has been adjusted to the computation-to-communication ratio most efficient on given hardware.

1 Introduction

What is the problem being solved Designing an OS that exposes no hardware details is a special challenge because an OS is intimately tied to the hardware. In order for parallel software to enjoy the commercial advantages that serial software has benefitted from, it needs to be equally hardware agnostic.

Such an OS must provide all the services required by programs and users. These include the ability to run a program, to install programs and data, to access data from a program, to communicate between programs, from programs to other machines, and from programs to hardware, and a security mechanism.

Why does the problem matter Parallel programming that is hardware-independent is increasing in importance due to the major shift in silicon-chip processors, which are transitioning from a single thread on a chip to an exponentially increasing number of threads on each successive generation of chip. Hardware independent programs require an OS that does not expose hardware details to the application.

Also, development costs for scientific and engineering codes on supercomputers is a limiting factor to scientific and engineering progress. These costs would be significanly reduced if the same code could be written and debugged once, on low-cost machines, then run, with high efficiency, on any current or future super-computer.

What are the fundamentals of the problem An OS presents, to an application, the interface to the "universe". Everything outside of an application is invoked through the OS's abstractions.

For the OS to present a HW-implication-free interface to an application, its abstractions must be HW-implication-free. Not only must they carry no implications about the hardware under the OS interface, but also not involve any OS-implementation related information.

What is hard about this problem Such an OS interface is difficult to define because HW implications arise in many places:

1. Implied by the format of primitive data types in data
2. Implied by location-information attached to names
3. Implied by the semantics of the computation model that code is in terms of
4. Embedded within code via OS commands, HW commands, and location-containing names

What is needed to solve the problem Isolating an application from these sources of HW implications requires:

1. A data-format boundary around all the applications, only allowing data to cross the boundary by passing through a data-format translator.
2. Separating pure names from both location information and meta-information. Only the OS's name-space mechanism needs to know location information. Conversely, only applications and people need the meta-information – to search for desired data's pure name.
3. An intermediate format for programs that is HW independent and allows translators that make high performance formats on particular HW.
4. An OS interface whose commands don't contain HW implications

What is unique about the proposed solution that enables its benefits

1) the four abstractions used to model the universe for the app, 2) the separating of pure name from attached information, 3) adaptors, 4) translation-at-install, and 5) a "dynamic" circuit intermediate format.

What are some details of each of these

1) The abstractions are data, processor, specification, and name-space. Processors live in a name-space and use it to communicate with other processors. Specifications are special data that say how to make a processor and then are part of the created processor.

In this scheme, programs are specifications, and a running program is a processor. Data-files are processors that contain the data. The data is accessed by a program by using the name-space mechanism to send commands to the data-containing processor and receive responses from it.

What used to be known as a "machine" or "server" is now the name-space of an OS-processor. In current widely used OSes, the OS-processor is transient. It is a processor instance created from the OS's compiled code by the boot sequence. Processors, which are running programs, communicate from one OS-instance's name-space with processors in another OS-instance's name-space, or "machine", by using a containing name-space such as a network.

A number of persistent OS-service-providing processors are part of every instance of the proposed OS interface. These processors are the means to create new processor instances (ie, run a program), provide GUI interfaces, search for names of desired processors, and so on.

To make these abstractions work in practice, instances are made persistent for the proposed OS. A rarely-performed step is undertaken in which a processor-instance of the OS is created. This OS-instance is then persisted, so that its name-space remains across power-cycles, and persistent-processors such as data-processors and OS-service processors remain.

2) The separating of pure name from meta-data and from name-space-mechanism-specific-data is accomplished by the processor-symbol and the name-discovery service. Once a program has a name of a processor, it communicates with that processor via a processor-symbol. These are placed into the code by the programmer. The program first connects the symbol to the desired processor by giving the symbol the processor's name. Then the program hands commands to the symbol. They are communicated by the OS implementation to the connected processor, which sends back a response, which then comes out of the processor-symbol.

Meta-data is used to find the pure-name of another processor that is inside the same OS instance (or the name of an external processor). That pure-name is given to the processor-symbol, which is the program's interface to the name-space mechanism. The OS implementation does something HW-specific to connect the processor-symbol to the named processor.

This keeps all the information attached to the name that has to do with details of the implemented name-space mechanism, hidden from the application. That information is created at install time and at creation-time, so the application doesn't need to know the contents, just to be able to invoke it. The search process, using the meta-data, is how the application finds the pure name that invokes the name-space-mech-specific info. The pure name is analogous to an interface, that sits between the meta-info and the location-info.

3) Adaptors go with the "data boundary" concept. Whenever data enters or leaves the OS instance's name-space, it crosses a data-format boundary. A processor that lives outside the OS instance has its own data formats. Inside the OS instance, internal data formats are used.

At the boundary, the data must cross while in the standard interchange format. An adaptor turns external-format data into the interchange format, and vice-versa. An adaptor is invoked either automatically, by the processor-symbol, or explicitly by a cast statement.

Data can enter a program as raw bytes, but it is treated as just that, an array of raw bytes. Those bytes can be manipulated with logical bit-wise and byte-wise operations, but all results in which an operand is a bit or byte type, the result is also. The only way to use such data as anything else is to either transform it bit-by-bit, using logical operation results to direct "if" statements, or to cast the data, specifying an adaptor.

This enforces the data-format boundary in all cases, while still allowing low-level bit-wise and byte-wise operations.

4) Translation-at-install means that to get inside an OS instance, data has to go through an installation process. This process creates a persistent processor to contain the data, creates the name-space-mechanism-specific information to locate the processor, creates the pure-name to specify the processor, and inserts the meta-information about that data into the name-discovery service, allowing later searches for the pure-name of the data.

The installation process uses adaptors to transform all external-format data to interchange-format data and from there translates to internal-format data.

The installation process also translates code from intermediate-format code to internal, HW-specific, format code. This involves dividing up the code into execution-units such that the complexity of the units fits the latency and BW of the name-space-mechanism (network), and the computation speed of the processors, given the expected data sizes (known from bundled profile info). And then, translating each of those execution-units into a single atomically-executed serial piece of code in terms of the computation model of the underlying OS (ie, machine code).

This install process removes all embedded HW-implying information, and creates any needed HW-specific information, hiding it underneath the adapator abstraction, the intermediate format, and the pure-name mechanism

5) **Dynamic circuit** means that the number of processors is un-specified. Each operation has a processor created to perform that operation on one piece of incoming data. When complete, the processor disappears from existence. It is unspecified the order or overlap of the existence of these processors. Only causality of data is specified, meaning data must leave one operation before it can begin any following operations.

   Hand in hand with this, control data is bundled with working data. Each operation only has local variables that exist for the duration of a single atomic operation execution. Thus, no side-effects exist that could expose ordering.

   Also, when two or more separate pieces of data are combined in an operation, the format includes guards that state a boolean on the contents of the data-pieces. When the contents of a proposed set passes the boolean, then the set has the function performed on it. This creates a partial ordering between separate tasks, where the bundle of control plus work data represents a task, the work data is the result-in-progress of the task, and the control data is bookeeping that implies how-far-along is the task.

   Finally, the intermediate format has a mechanism by which the scheduling process on the hardware communicates directly to the running program, answering queries as to how many pieces given data should be broken into. Together, these aspects of the intermediate format allow the same program to be translated at install-time to fit the hardware, and allow the running program to remain ignorant of the implementation details of the scheduling process but still divide up its data into hardware-specific sizes.

What results need to be shown To be completely convincing, we need to show the proposed OS interface implemented on a laptop, a network of workstations, and a dedicated NUMA supercomputer. We need to show at least two programs, that use all of the OS's services, run on all three configurations, from the same distribution bundles. The name-discovery service, the authentication service and the GUI maker are the most complex from the application's viewpoint and so need the most attention. Ideally, a "real" program is run across those platforms from a single distribution bundle.

It would be especially nice to show what it's going to be like to use a GUI to communicate with a person, including the process of finding the person and creating a GUI on the screen they are viewing. Similar examples of using the name-discovery service and the authentication service would be nice, especially to illustrate how the proposed authentication service fits over the top of Windows and Linux style access-control-lists, as well as, say trust domains.

What results are given in the paper The paper shows proof-of-concept results for one implementation of the OS with one program running under it on a laptop and on a network of workstations. We state only which sub-set of the services we implemented, what the program does to exercise those services and state that the program runs correctly.

What is the justification for the results left out This paper uses its allotted space to state the concepts and details of the proposed OS interface. Those have to be stated somewhere, and giving sufficient detail to be understandable uses up the allotted space. Implementation detail is given in a paper[?] describing the run-time system, which implements many parts of the proposed OS interface on the hardware platforms. Future work will fully implement the more complex portions of the interface.

2 Description

Here we describe the application-visible portions of the proposed OS interface. These include the notion of persistent OS instances, the processor-symbol by which OS services are invoked, each of the built-in processors that together provide many of the OS services, and name-spaces. We give details, in turn, for:

1. OS Instance. – Persistent "run" of the code that implements the OS interface
2. Processor-symbol – The application-visible communication mechanism
3. Install processor – Invoked to install data, including data that is a program
4. Creator processor – Creates new processor instances inside the OS instance
5. UI-maker processor – Creates processors through which programs interact with user
6. Name-disovery processor – Programs use to search meta-info to find a desired processor
7. External Listener – Connects to processors outside the OS instance
8. Authentication processor – Checks if authentication passes requirements for a command
9. Name spaces – The thing that defines inside a processor vs outside it

We do not discuss the intermediate format, the computation model, or the run-time system as these are described fully in other papers[?][?][?].

In the section following this one, we describe the data-format boundary, adaptors, and the ways in which applications are isolated from HW implications.

2.1 OS Instances

An OS instance is persistent. An analogous idea is the "suspend to disk" state of a running Linux session. It consists of all of the internal state of the processor that is a run of the OS implementation.

The proposed OS takes a slightly different view of "booting up". In most, traditional, OSes, each power-on of hardware generates a new instance of the OS. In constrast, the proposed OS interface has persistent instances. The power-on of the hardware only allows the state of an already-existing instance to evolve. It loads in a portion of the persistent state and re-creates the built-in processors. Commands may then be given to the built-in processors, users may log in to the OS instance, and so on.

Data files are persistent processors that are part of the OS instance. They are created by installing the data into an OS instance, which creates a new persistent processor that holds the data and can be asked to give the data, or change the data. A program can "discover" the presence of the data-processor, open a link to it, then send read and write commands to it.

The persistent state of an OS instance includes any data that has been installed, any persistent data that has been created inside the OS instance, the OS instance's name-spaces, and any persistent bookeeping that the OS instance keeps.

Upon power-up of the hardware, the OS instance's persistent processors become active. How this is done is implementation-specific. Their state might be loaded in to main memory, or just some information about them.

A particular OS instance has physical resources assigned to it. For example, a machine running Unix would have a process that is a run of the run-time-system. The run-time system code implements essential parts of the OS interface, notably the name-space, so it is the gateway to all OS services. That run-time system process equals machine resources assigned to an OS instance.

An OS instance is like a huge program-image, with most of it on disk. An analog is that the OS is "run" once, most of it is swapped out to disk, and a small portion is re-created in main memory each time processing resources are assigned to it, for example, when the hardware is powered up. Thus, the state of this single, persistent, "run" changes, on disk, over time. This "run" remains after the hardware is powered off (processor resources are taken away), part of it is re-created when the hardware is turned back on, and it only changes while the hardware is powered up. It is possible to assign only a portion of given hardware to a given OS instance, for example, assigning one process on a Unix machine to a given OS instance. Thus the same powered up hardware can be causing changes in multiple OS Instances, or running programs under a different OS "simultaneously" with running programs under the proposed OS within its assigned process.

In related papers, the name "Virtual Server" is used to denote "OS Instance".

2.2 Processor Unit

2.2.1 The run-time system

The run-time system implements several things: the processor-unit abstraction, the OS instance's name-space mechanism, part of the creator processor, and the scheduling process for the executable modules generated by the install-processor.

2.2.2 The processor-unit abstraction

The processor-unit symbol is the means by which one processor communications with other processors in the OS instance's name-space. As such it is at the heart of the OS interface, being the application's gateway to all the built-in processors that provide the rest of the OS's services.

A processor-unit symbol is placed in a program, then during a run, the program presents a name to the symbol. This invokes the OS's implementation of the processor-unit. The implementation looks up the name in the OS instance's name-space, gets hardware-specific communication information, and uses that information to connect the processor-unit symbol to the named processor.

If the named processor is a running program, then the commands come into that running program via "input-pin" symbols and responses are sent out from the named-processor via "output-pin" symbols. Pins are the "dual" of a processor-unit. The processor-unit ports in one running program are hooked up to corresponding pins in the named running program. A program may include a "start" input-pin which causes execution to begin without any pins receiving data, and a "finished" output-pin (data sent to this is analogous to an exit code).

To illustrate the difference between the processor-unit symbol and pin symbols, consider the case when a running instance of program A causes the creation of a running instance of program B. The running instance of program A gives the name of program B to a processor-unit symbol. This invokes the OS to: 1) look up the name, and see that it is a specification, 2) create a new processor from that specification (which is running program B), and 3) connect the ports of the processor-unit symbol in running program A to the corresponding pin symbols in running program B. In the source code, the processor-unit symbol in program A will have a port with the same name and data-structure type as pin symbols in program B. The OS connects processor-unit ports to pins with the same name and data-structure type.

2.3 Install processor

2.3.1 Programs as processor-specifications and the distribution format

Programs are referred to in this OS as processor-specifications. One particular run of a program is referred to as one particular processor, which was made according to the processor-specification (according to the program).

Programs are presented to the install processor in a standard distribution bundle. A bundle has the program-code, data that the program uses, and install information such as adaptors, security requests, and names to give to files. The programs are stated in terms of the CodeTime Intermediate Format, CTIF. The semantics of CTIF are given by the CodeTime computation model's operational semantics[?].

2.3.2 What the install processor does

The install processor translates from the intermediate format into an implementation-specific internal format. The internal format is defined by the run-time system. For example, one run-time has been implemented which accepts java class-files, each of which implements the same interface. The install processor generates these class files from the CTIF code it is given. Each class file can be considered as one command to the processor that the run-time system spec defines. Another run-time system on different hardware might accept a DLL, or a collection of object files (the implementations of the run-time and the install-processor are fully aware of all hardware details and the native OS of the hardware, but that is all hidden from the application by CTIF).

2.3.3 Installing data into an OS instance

The proposed OS interface has no concept of a data file, it has only the processor abstraction. So, data files are turned into persistent processors by the install processor. To do this, data is "installed" into an OS instance.

When data is installed into an OS instance, it becomes a persistent active processor. The OS implementation comes with a built-in specification of file-processors, which is used to create the new, persistent, file-processor instance. Also during installation, the name of the file-processor is put into two places, the name-discovery processor and the OS instance's internal name-space mechanism.

In the name-discovery processor, the name has attached meta-information supplied by the distribution bundle that contained the file (or supplied by a user or a program). The meta-information includes things like the name of the bundle the file came in, the names of programs that understand its format, the name of its format (analogous to the three-letter extension on some file-systems), and so on.

When the new persistent file-processor's name is placed into the OS's internal name-space mechanism, hardware-specific information is generated and attached. The hardware-specific information may include URL, file-system, path, and so on. The hardware-specific information is retrieved and used when a program gives a name to a processor-unit. The OS instance internally looks up the name to retrieve the hardware-specific information it needs (this activity is performed by the run-time system, which implements the processor-unit abstraction and the OS instance's internal name-space mechanism).

2.3.4 Processor specifications (programs)

An installed processor specification is a persistent processor, just like any other data-file. It is only distinguished by meta-information. When the creator processor is asked to create a processor from a named specification, the creator processor uses the name to connect to the persistent processor that holds the specification. It then sends a request for the specification data.

When a program is found via the name discovery processor, what is found is actually the name of the persistent processor that holds the specification data.

2.4 Creator processor

This takes an already-installed processor-specification and creates a processor from it. Such a created processor is analogous to a process on other OSes. The created processor might be persistent, or might be transitory. For example, the install processor calls the creator processor to make a persistent processor as part of the install of a data-file.

The implementation of the creator processor is "inside" the run-time system. Whatever hardware-specific thing the run-time does to initialize and begin the run of a program is the implementation of the creator processor.

Sending the name of a processor-spec to the creator processor is what causes the run-time to begin execution of that program. The run-time uses its own name-space implementation to lookup the name of the processor-spec, which retrieves the set of internal-format modules generated by the install processor.

When the program was installed, one of the modules that it was translated into was an initialization module. The run-time system uses this to create its internal representation of the processor instance.

The creator-processor, as implemented by the run-time, also makes a name for the new processor instance and registers it in the internal name-space mechanism, as well as sending the name plus meta-info to the name-discovery processor.

The creator processor may be invoked by the install processor during installation of a data-file, or by a processor-unit abstraction. When a running program (a processor) hands a name of a processor-spec to one of its processor-unit symbols, the implementation of the processor-unit invokes the creator-processor to create a new instance of that spec. The processor-unit symbol implementation then attaches the newly created processor to the processor-unit symbol. Any data-containers that the program hands to the processor-unit symbol are communicated to the new instance.

The run-time, after creating a new processor-instance, then waits until some processor-unit symbol attached to that instance is handed data. The run-time determines which modules contain the pin-symbol that the processor-unit port is connected to. It then sets about scheduling execution of those modules, on the data, on specific processors. The completion of those modules generates more data destined for other modules, and so on. The generation of data and scheduling of it continues until data is handed to the "finished" symbol in the program, which causes the processor instance to be destroyed (note, however, that no resources other than memory are taken up by leaving the processor instance in existence, by not sending data to the finished-symbol).

The creator processor also destroys instances. One processor may send a request to the creator processor to have another processor destroyed. The authentications owned by the requesting processor must pass the requirements owned by the targeted-for-destruction processor for the destruction operation. If the authentication passes the requirements for the destruction operation, then the run-time, acting as the creator-processor, internally eliminates whatever it had to represent the now-destroyed processor. This is analogous to a shell created from a super-user login being able to destroy a process.

2.5 UI-maker processor

The UI-maker is really part of the creator-processor, but usually creates processors on external machines and so is treated as separate. It creates "user interface" processors, of several types including graphical ones. The commands accepted by each type of UI processor are defined as part of the platform standard. One such type may have commands equivalent to an X-Window, or MFC, or Java Swing. For games, a processor type that takes commands equivalent to DirectX, or OpenGL, may be implemented, thereby reducing the need for direct hardware interaction.

For example, one type of UI processor is a window that can be given commands to populate it with widgets and contents, as well as to decorate it. The UI processor, in turn, sends commands containing mouse movements, clicks, key-strokes and so on.

A UI must be connected to a specific physical machine. The normal way to do this is to make the machine an external processor that connects through the external listener. The UI processor in this case is an external processor that is outside the OS instance. This is because it is allowed for the external listener to know about physical properties and particular addresses in particular communication protocols of external machines.

External machines must be registered with the external listener, which causes the machine-name plus meta-information about the machine to be registered with the name-discovery processor. The meta-information might include physical location, or the name of the user sitting at that machine. A program can then discover an external machine that has the desired properties (meta-information), such as name of person. The program then hands the name of the external machine to the UI-maker, along with the type of UI desired, and gets back the name of the created UI processor.

It is straight forward to implement this with a window system such as X-Windows which has a server that can be sent commands from external machines to create and manipulate windows. Other situations may require a "stub" to be running on the external machine which interacts with the external listener and can accept the command from the UI-maker to create a new window.

2.6 Name Discovery Processor

The name discovery processor is used to search for the names of processor instances. All of the built-in processors have a fixed name in every instance of every implementation of the OS interface. However, things such as data-files, CD-ROM drives, printers, and displays in particular physical locations are specific to each, persistent, OS instance. They must be searched for by a program. The result from a search sequence is the name of processor that is then given to a processor-unit to establish communication.

The search is performed by giving search criteria to the name discovery processor, which responds with a list of names of processors whose meta-data matches the criteria.

The meta-information has fixed categories that are defined as part of the OS interface definition.

The entry in the name discovery processor, including meta-data, is created when the processor is created. Creation causes both the name of the newly created processor to be added to the OS instance's internal name-space, and the name to be given to the name discovery processor along with meta-information about the name.

The detailed protocol for performing a search session is defined as part of the OS interface definition. It is expected that the name discovery processor will be implemented with a relational database, and so a search session has a resemblance to a SQL session with a standard database.

Most names will be quite easy to find, especially the names of data-files which are packaged together with a program. Thus, most interactions with the name discovery processor are straight forward, and amount to simply using a wild-card to fill in the OS instance specific base portion of a name.

2.7 External Listener

The external listener is the means by which processors that do not exist in the OS instance's internal-name-space, ie are external processors, can communicate with processors that are internal to the OS instance. For example, if a remote user wants to log in to the OS instance and run a program, the external listener is the only processor capable of them initiating communication with. The external listener forces them to perform the authentication process, after which it causes a new shell-processor to be created. The shell-processor will own the authentication that the remote user completed. Any command the shell attempts to perform, such as running a program, will only be allowed if the authentication passes the requirements for that command.

The external listener performs a registration process for external processors and external machines. This registration adds the external processor/machine to the OS instance's external name-space. It creates a generic name for that external processor/machine and places it into both the name discovery processor, and the OS instance's external name-space. Also as part of registering the external server, the data-structures that may be communicated to and from that server must be specified, including the lexical information needed to parse incoming data.

Any primitive data types within these structures must be in the standard interchange format (see section 3.2.1). If they are not, an adaptor program must be registered with the External Listener processor.

When data is sent to or from the external processor, it passes through the external listener processor. The external listener parses raw data coming-in into internal-format data-structures which it then passes on to the connected internal processor. While parsing, the external listener hands primitive data types to an instance of the adaptor registered to the external processor for translation to the standard format. It then translates from standard format to internal format. This is how the OS enforces independence from HW details of data-formats of external processors.

2.8 Authentication Processor

2.8.1 Security Model

The security model is abstract. Three data-elements are involved in security decisions:

1. Authentication – black box created by the authentication processor
2. Requirements – black box created by the authentication processor
3. Operation-type – name of set of operations

The first two are black-boxes. They cannot be modified or even looked inside of. They can only be created by the authentication processor. However, processors may pass them around by name.

To use them, a processor sends a request to the authentication processor. In the request is the operation-type, an authentication and a requirements. The authentication processor responds with wether the authentication passes the requirements for that operation-type.

Each processor decides for itself whether it will perform a command requested by another processor. When a processor wishes to verify that the requestor passes the requirements for the requested operation, it sends a requirements-check message to the authentication processor.

This framework leaves open the choice of security mechanism implemented. Access Control Lists fit into this framework, as do Capabilities, and even trust domains.

2.8.2 Authentications, requirements and using the authentication processor

The authentication processor creates authentications by performing an implementation-specific authentication sequence. A typical sequence would be a login and password session.

Authentications and requirements are owned by processors. Ownership is aquired during creation of the processor. Either can be sent, by name, to another processor, which then has the received authentication or requirement by proxy.

Authentications and requirements are used by processors for security checks. A processor that desires a security check sends an authentication, a requirements, and an operation-type to the authentication processor, which responds with a pass or no-pass.

Note that the response from the authentication processor has no intrinsic value. Passing the response along to other processors achieves nothing. Only the requesting processor gives weight to the authentication processor's response. Also, the authentication processor has a fixed name in every implementation of the OS interface. The implementation includes the name-space mechanism. Thus, portions of the OS implementation code would have to be changed in order to spoof a response from the authentication processor (assuming no bugs in the OS implementation).

When an authentication or a requirements is sent, it is marked as either owned by the sender or sent by proxy. A receiving processor performs a keyword defined in the intermediate format to check the mark. The implementation of the processor-unit performs the marking, so it cannot be counterfieted. The sending processor places the names of the authentications and requirements into the structure that it sends. When the structure is handed to the processor-unit through which it is sent, the processor-unit marks whether the named authentications and requirements were owned by the sender or not.

It is up to each processor to decide how it wants to use proxy-authentications and proxy-requirements. The authentication processor gives the same answer whether it is given proxy-versions or owned-versions. However, the authentication or requirements that comes out of a processor-unit or a pin has a special keyword that can be performed on it which returns true iff the authentication or requirements was owned by the sender. The processor receiving it can then decide whether it will accept a proxy or not.

In addition, when one processor creates a connection to another, it may attach a default authentication to the connection. After establishing the connection with a default, every subsequent communication automatically carries the authentication the connection was created with (at least logically, if not physically).

The requirements a processor is created with are generated in an implementation-specific way. The means to generate the requirements given to a specific processor instance being created is stored with the specification (program) the processor instance is being created from. This means is implementation-specific and is attached to the specification when the specification is installed into the OS instance. The means that is stored with the spec may be a program, or it may be simply a data-set (such as owner-group-world permissions).

Installing a specification (program) requires stating the operation-types a running instance of the specification can ask the authentication processor to check requirements on. Installing also requires some implementation-specific steps to create the means of generating the requirements that is stored with the newly installed spec. The implementation of the install processor may automate creating the means and stating the operation-types.

When a request is sent to the creator processor to create a new processor instance, the authentications that the new processor is to own must be part of the request. Any authentication that the requesting processor owns may be given to the created processor. Alternatively, the requesting processor can have the creator processor obtain a brand new authentication from the authentication processor. The new authentication would be the result of the authentication processor engaging in the authentication sequence with a specified source processor (how to guarantee that a given communication really does come from the believed source is implementation specific, for example by the IP address, or the SSH client's key).

That is how a remote user gains a shell that has their particular authentication that resulted from the login plus password sequence. The external listener was asked for some kind of connection by an external machine. Each implementation chooses which kinds of connections it recognises as valid sources: TCP, FTP, SSH, and so on. The external listener then asks the processor creator to create the kind of shell appropriate to the kind of source, and to get the authentication for that shell from the authentication processor.

After this, the new shell will send the newly created and now owned by it, authentication when it requests actions from other processors inside the OS instance. Thus, it will be the newly created authentication that is used by the name-discovery processor to decide which names the shell is allowed to see (by storing the proxy-requirements with each name, and asking the authentication processor whether the new authentication passes the proxy-requirements for the viewing operation), and by the processor-unit implementation to decide if a requested connection is allowed, and by the creation processor to decide if the requested run of a particular program is allowed, and so on.

2.9 Name Spaces

A name-space is the mechanism by which two processors achieve communication. The implementation of the name-space is the physical thing that gets information from one processor to the other. The name of the destination processor is what is used to get the information to that destination processor. Any communication mechanism, such as a network, is considered, in this OS's parlance, to be an implementation of a name-space.

An OS instance has an internal, persistent, name-space, and it exists within an external name space. The internal name-space implementation is part of the run-time system. The external name-space may be physically any number of different communication mechanisms, including main memory, a LAN, and the internet. The external listener translates pure names into external location-information.

The two name-spaces define "inside the OS instance" vs "outside the OS instance". The name of any processor known to the internal name-space is "inside the OS instance". This includes persistent data-file processors and persistent OS-service processors. All other processors are outside the OS instance. Only outside processors that have been registered with the external listener may be communicated with.

The same physical machine can have processors that appear in both name-spaces. For example when the proposed OS is implemented on top of Linux, other processes in the same Linux instance may act as external processors and connect to the external listener. Thus, two different programs run on the same physical machine, one is under Linux, the other is under an instance of the proposed OS. Both are within Linux processes, but it is the proposed OS instance's name-space that defines which is external and which is internal to that OS instance.

3 Enforcing Genericity

3.1 Data-format boundary

In order to remain generic to all hardware, a data-boundary needs to be established around an OS instance, and all hardware details embedded in data that crosses this boundary be abstracted. This presents a challenge when raw data is communicated to or from a program, or data is embedded within a program (for example, strings that represent file paths, or constants with a large number of digits). Data may embed a number of hardware details, including:

- Communication format (including file-system format, encryption, and so on)
- Instruction set (executable files)
- Higher-order hardware details: machine URL, drive number, device-specific commands (graphics card commands, printer commands)
- Primitive data-type format: character, integer, floating point number (8-bit ASCII vs 16-bit uni-code, 32bit int vs 64bit int, big-endian vs little, single precision vs double)

3.2 Genericising an application's view of hardware details in data

An application needs to be able to interact with other internal processors and with external processors and yet not be exposed to the hardware details embedded in the data communicated. This is accomplished by the proposed OS in the following ways:

1. Communication format is abstracted by providing the generic processor-unit abstraction, through which all communication to and from a program takes place. The actual details of protocol, file-system, in-flight encryption, and so on are handled by the implementation of the processor-unit abstraction. Each hardware platform has its own processor-unit implementation.
2. Instruction set details are abstracted by the intermediate format[?]. All programs runnable within the proposed OS must be communicated to the OS in this format. The proposed OS provides no mechanism to run images pre-compiled for specific hardware.
3. Higher-order details are abstracted by either the processor-unit abstraction or one of the built-in processors. For example, machine URL, drive number, and file-system path are all abstracted by the name-discovery processor. A program uses the name-discovery processor to search for what it wants. A program receives the pure name of the processor it found, and uses that to communicate.

   The program gives the pure name to a processor-unit symbol, which causes the OS to connect the symbol to the desired processor-instance. The processor-unit implementation translates the pure name into hardware-specific URL, path, and so on. The OS's name-discovery processor allows the program to have only application-related search-information.

   Sometimes a programmer conciously wants to include hardware-specific commands to interact with particular hardware. For example, a program may control machines in a factory, or it may want to manipulate the hardware registers on a PCI card. It forms the commands as an array of characters, numbers, or bits. It then hands the array to a processor-unit connected to the hardware. The hardware must have previously been registered with the external listener, at which time adaptors were specified that translate the program-provided array to the bit-stream the hardware requires.

In this way, a program can use the name-discovery service to find the hardware, via its meta-information, then connect to it, no matter what kind of bus it might be connected to, or which physical box the card might be in (meta-info allows searching for a specific box, if needed), and so on. This provides maximal isolation of program from hardware.

4. Primitive data-format is abstracted by adaptors. Adaptors translate between the external data-format and the standard data-format. All data crossing the OS instance boundary must be in the standard format. This standard format states bit lengths, bit meanings, and so on. For example, IEEE-754 defines the standard format for floating point numbers (and also the computation model for floating point numbers).

   When an external server is registered with the OS instance, in addition to a name for that external server being created and placed into both the name discovery processor, and the OS instance's external name-space, the data-structures that may be communicated to and from that external processor are also registered. This includes the lexical information needed to parse incoming data. If the data contains primitive data types that parse to a format other than the standard, then an adaptor must also be registered.

   When a primitive data type is then parsed, it is handed to an instance of the adaptor for translation to the standard format. Internally, the OS instance may then translate the data to an internal format that that OS instance's processors use.

   When data on some communication medium such as CD-ROM is placed into a reader, it must be one of: 1) in a standard format that the OS has built-in data-structure, parsing, and adaptor information for, 2) packaged with the data structures, parsing information, and adaptor information, or else 3) the data-type, parsing, and adaptor information must be supplied by the person inserting the CD-ROM. The data can then be either communicated to a program, during which the CD-resident file acts as a registered external processor, or it can be installed into the OS instance. Either way, the data is parsed into data-structures and translated into the standard format then to the OS instance's internal format as it crosses the OS instance boundary.

3.2.1 Adaptors

An OS instance can be thought of as having a data-format boundary. No data crosses this boundary without being parsed to or from an explicitly declared format. Incoming data is parsed from raw bytes into data-structures, and vice-versa. If the incoming raw bytes are not in the OS's standard format, then an adaptor must be provided which performs the parsing and then translates to the standard format. If the adaptor is not one of the OS's standard ones, then it is supplied by the external entity.

The adaptor translates the data to an OS-defined intermediate data-type format.

- Integers may be: fixed-length or else infinite precision. Fixed length integers are 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit, all with least significant byte at the lowest address.
- Floating point may be: 32-bit, 64-bit, 128-bit, 256-bit IEEE-754
- Characters are either 8-bit ASCII or 16-bit ISO

In addition, the OS defines protocol-adaptors which can be "stacked" to make higher-level adaptors. In this way, layers of protocol can be stripped off and re-applied.

For example, consider a web-server. Data coming in is packaged in the physical packet protocol, then in the IP packet protocol, then in the TCP connection protocol, then in a primitive data-type format, then inside the HTTP protocol, then inside the HTML protocol. For the web-server to understand the data itself, it must be converted to internal-format after coming out of TCP, then all the protocol information must be converted to data-structures. A stack of adaptors can be provided which do this, each adaptor translating one protocol or format.

The higher-level adaptors that are applied "above", or after, the data-type adaptor, such as the HTTP protocol adaptor, are all normal programs. They are distributed in the OS's intermediate format, so they need be written and compiled-from-source only once, and can then be used on every instance of the OS.

Data cannot change type inside a running program without a cast statement. This requirement ensures that no path exists for data to enter a program as an array of bytes, then somehow be used as a data-structure without passing through an adaptor. The cast statement can be made on an array of bytes as long as the cast specifies the data-structure and an adaptor. Likewise, a data-structure can be turned into an array of bytes by a cast, which can then sent to an external processor, but this is deprecated in favor of registering the adaptor used in the cast with the external listener and having it automatically applied by the processor-unit.

When data on some communication medium such as CD-ROM is placed into a reader, it must be packaged in a standard format that includes the data structure, parsing information, and adaptor information. The data can then be either communicated to a program, during which the CD-resident file acts as a registered external processor, or it can be installed into the OS instance. Either way, the data is parsed into data-structures and translated into the standard format then to the OS instance's internal format as it crosses the OS instance boundary.

4 Results

We have implemented major portions of the proposed OS interface. The computation model is implemented by a combination of hand-compiling and a Java-based run-time system[?]. The name-space mechanism, including the processor-symbol, is implemented as part of the run-time system. The creator service is implemented as part of the run-time system. A primitive name-discovery service is implemented in Java, as is a UI-maker that connects through a stub to locally run Java "controller plus view" applications. The installation process is performed mainly by hand, but installations are persistent. The persistent OS instances are implemented in part by the hand-install process, and in part by the run-time system. One standard adaptor is implemented as part of the install process, but custom adaptors are not nor is the use of adaptors in casts inside a program. A rudimentary authentication service is implemented, as is a bare-bones external-listener that only allows connection through a stub to external applications, such as those running a GUI that the UI-maker "creates".

We implemented this system for a network of 4 Xeon 3Ghz workstations running Linux, connected by 100 mbit ethernet. The same implementation was developed and runs on a 1.5GHz Centrino laptop. The application we ran is a matrix multiply program that takes a total of 10 matrices off disk, multiplies each combination, and writes the results. This application uses the name-discovery service to find the matrices, the name-space mechanism and the processor-symbol to connect to the data-processors that contain the matrices, the creator-processor to make new data-processors to hold the result matrices, and the UI-maker to make displays that show the contents of the original and the result matrices. The application runs correctly and displays the results on the workstation from which the user logged in.

5 Conclusion

This paper has described an OS interface which should be easy to implement on top of existing OSes that provides a generic interface to application programs. The programs are insulated from hardware-specific details such as number of processors, instruction sets of processors, URL of machines it is running on, paths, file-system, or even primitive data format.

This genericity is accomplished by the use of an intermediate program format, the use of the processor abstraction for communication, the inclusion of built-in processors for OS services, and the use of adaptors to make data-formats generic.

Bibliography