# A Hardware-Independent Parallel Operating System Abstraction Layer

by Sean Halle

Website: ctos.sourceforge.net
Email: seanhalle@yahoo.com

*January 15, 2007*

**Abstract**

Having a single operating system interface has been a boon in many segments of industry and society. It would be nice to have a single operating system interface in the age of parallel hardware. Such an interface would enable source code to be written once, then automatically specialized to each underlying hardware platform. The single OS interface means that all services that an application requires are obtained via a single set of commands, across languages and across hardware platforms. The single OS interface also means that low level hardware issues such as endianness, sizes of primitive data types, file-system details, and even particular paths are hidden from the application.

Designing an OS that exposes no hardware details is a special challenge. We propose to use an abstraction layer over the top of native OSs that separates the interface to OS services from the implementation of those services. In our abstraction layer we adopt the abstraction that everything is a processor. On one side, this abstraction is simple for source code to use and meets all the needs of applications, while on the other side the abstraction is easy to implement on an especially wide array of hardware configurations, as a layer above the native OS.

The proposed OS interface uses four elements to achieve the separation: 1) the abstraction that everything is a processor; 2) using pure names free from implied information; 3) adaptors that translate data formats; and 4) automated specialization that translates a single source to multiple machine specific forms.

## 1 Introduction

An OS presents, to an application, the interface to the "universe". Everything outside of an application is invoked through the OS interface. Current OSs have subtle intrusions of hardware-exposing details into their interface designs. Such HW implications happen in:

1. Machine binaries, as opposed to more abstract representations of programs
2. Incoming data streams whose bit format is defined outside the OS
3. Names, embedded in data or code, that imply location-information or meta-information
4. The computation model that code is in terms of (may imply a Hardware configuration)
5. OS commands or HW commands that state where or how to perform a service (where and how are in terms of particular machine details)

Isolating an application from these sources of HW implications requires:

1. An OS interface whose commands rest on a sufficiently abstract machine model.

2. Separating 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. A boundary around all data and programs. (Crossing the OS boundary translates between internal formats plus computation models and external formats plus computation models.)

# 2 Description

**Some details of how the proposed OS abstraction layer achieves isolation:**

**1) The processor abstraction** As seen from within an instance of the proposed OS, everything is a processor: OS services are obtained by interacting with processors; all data resides inside processors; and programs are specifications used to create processors.

In fact, "The OS", as seen by an application, is itself a processor. The OS-processor holds within it all file-processors, all processors that are running programs, and special OS-service-providing processors. The OS-service processors include: one that creates new processors (ie, runs a program); one that creates GUI interface processors; one to search for names of desired processors (such as "files"), and so on.

The OS processor is persistent. It remains across power cycles of hardware. Hence, processors created inside an OS instance also persist.

For example, a "file" is a persistent processor that holds data and lives inside an OS instance. To find files, rather than use fixed names such as paths, instead a search mechanism is used. Meta-data about the file is registered with the search processor when the file is installed into the OS instance, then used when programs search for the file.

**2) Pure names** A pure name has no information attached to it. Pure names are often used in data-bases, where each record is given a meaningless number as its name. The record is retrieved via the number rather than information in the record. The number is found by doing a search about the data in the desired record. Likewise, in the proposed OS abstraction, desired data is found by searching for it. The search returns a "black box" that contains machine-specific information used to retrieve the data. The black-box is the same as the meaningless number in the data-base.

The OS abstraction has the concept of a name-space. A name-space is the mechanism that carries out communication. In the proposed OS interface, the name-space is explicitly represented via syntax in source code, either as a keyword or as a procedure name. The programmer sees the name-space syntactic entity as "a connection to a processor". The code specifies handing the syntactic-entity a name plus data, which causes the data to be delivered to the named processor.

Meta-data is used to find the pure name of a processor, via the use of an OS-provided search service. What would be a path in Linux is meta-data in the proposed OS abstraction. To open a file a running program connects to the search service, sends the meta-info and receives back a black-box that is the name of the desired file-processor. This approach makes finding specific data more flexible than using paths. For example, a specific version of a file from a specific creation data can be chosen. This approach also makes such path-like information hardware independent; a program doesn't have to distinguish between /usr/bin and /usr/local/bin, it just searches for the application's meta-name.

This keeps hidden all the information attached to the name that has to do with details of the name-space implementation, such as machine-names, file-systems, URLs, and so forth. That information is created at install time and at file creation-time, and is passed inside the black box. The application doesn't need to know the contents of the box, just to be able to invoke it.

**3) Data Adaptors** The proposed OS abstraction enforces a bounary around each OS instance. Data that enters or leaves the OS instance's name-space crosses the 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. It does this via a pair of adaptors, one from external-format to interchange-format, a second from interchange format to internal format. An adaptor is invoked either automatically, by the name-space mechanism, or explicitly by a cast statement.

Internally, data is exchanged between processors in formatted form. Data can approach an OS instance from the outside as raw bytes, but it is formatted by the internal adapator. If a program wishes to treat data as 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. To use such data as anything else it must be transformed bit-by-bit, using logical operation results to direct "if" statements, or else it must be cast, specifying an adaptor. This enforces the data-format boundary in all cases, while still allowing low-level bit-wise and byte-wise operations.

**4) Translating programs** The proposed OSs boundary also translates programs that cross. Programs that enter an OS instance must first go through translation. The translation is normally performed in batch mode by a distribution service, but may also be performed locally at install time. Batch translation takes a single source bundle and produces several executable images, each for a specific hardware platform. A given OS instance then connects to the distribution service, requests the program, and receives the executable translated to the hardware that OS instance is running on.

This enables dusty decks to be translated to new hardware automatically and distributed as soon as the new hardware is available. The programs must be re-installed into a given OS instance when the hardware under that OS instance changes, but the user-data and environment will remain unchanged, allowing seamless continuance of usage by end-users.

To be convincing on the viability of the proposed abstraction layer, we need to show implementations of it on a laptop, a network of workstations, and a dedicated NUMA supercomputer. We also need to show at least two programs, that use all of the OS's services, run on all three configurations, from a single source bundle. In the applications, we need to pay the most attention to the most complex OS services, which are the name-discovery service, the authentication service and the GUI maker. Ideally, a "real" program would be run across the platforms, from a single source bundle.

It would be especially nice to show what it's going to be like to use a GUI to communicate with a person. 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 we do show is 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 evaluate only a sub-set of the services, and state what the program does to exercise those services, showing that the program runs correctly.

Implementation detail is given in a paper[?] describing the run-time system, which implements many parts of the proposed OS interface. Future work will fully implement the more complex portions of the interface.

# 3 Details of the OS Abstraction

The proposed OS abstraction layer includes: the notion of persistent OS instances, the processor-syntax used to invoke OS services, and the processors that provide the application-visible OS services. They will be discussed in this section in this order:

1. OS Instance – A persistent processor that is a run of the OS code
2. Communication Syntax – The application syntax to invoke communication
3. Name space – The communication mechanism
4. Import-Export processor – Imports and Exports data, including programs
5. Creator processor – Creates new processors, which live inside the OS instance
6. User Interface – Uses an external Display processor for user-interaction
7. Name-discovery processor – Used to search for desired data and other processors
8. External Interface – To communicate with processors outside the OS instance
9. Authentication processor – The security mechanism

What isn't covered here is the automated translation and distribution, and the proof-of-concept run-time system that the results were collected on. These are described in other papers[?][?][?].

The adaptors and other ways applications are isolated from HW low level details is discussed in the section after this one.

## 3.1 OS Instances

An OS instance is persistent. An analogous idea is the "suspend to disk" state of a running Linux session. An OS instance consists of all of the internal state of the processor created from the OS implementation code. This contrasts to other OSes, which create a new instance each time the hardware is turned on.

Having a persistent OS instance is useful for things like the data-format boundary. Everything inside the OS instance is inside the data-format boundary. Data only has to be imported once, during which it is translated to the OS's internal format, then the data persists inside the OS instance. Likewise, programs are imported once, then persist inside the OS instance. Processors that have been created from programs also persist, waiting for input, until explicitly destroyed.

Because it is persistent, the proposed OS abstraction 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, for the proposed OS abstraction, power-on restores a portion of the persistent state. Once some kind of hardware is active beneath the OS instance, commands may be given to the processors, users may log in to the OS instance, and so on.

Data files are persistent processors inside 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 accept new data. A program searches for the processor holding the data it wants then sends read and write commands to it.

A particular OS instance is simply a collection of state. It can be stored on disk. The state changes while the OS instance is active, which is done by assigning processing resources to it. For example, on a machine running Linix natively, one would start a process that would animate an OS instance. The process would be a run of a Lixux program that implements the OS interface. Thus, all programs that run inside an instance of the proposed OS abstraction, are actually running inside that one Linux process. For a network of Linux workstations, the Linux process would be a run-time system that handles communication and scheduling.

## 3.2 Communication Syntax

The means by which one processor communicates with other processors is at the heart of the OS interface, being the application's gateway to the processors that provide the rest of the OS's services. Each language has its own variation on the syntax, but they are all equivalent to a processor symbol. The symbol is given a name plus data, and delivers the data to the named processor.

## 3.3 Import-Export processor

This performs translation on data contained inside processors. It is one of the mechanisms that enforces the boundary around an OS Instance. There are several kinds of processor that can be imported or exported.

### 3.3.1 Programs as processor-specifications, and importing them

Programs are imported into an OS Instance by using the import-export processor. A "program" under the proposed OS abstraction is quite different from what one normally thinks of. Under the proposed OS abstraction, a program is a collection of processors. Some of those processors contain source code, others are running processors created from some of that source.

A unit of source code is technically called a processor-specification under the proposed OS abstraction. Running a unit of code is the act of creating a processor from that code. But a program can have a mix of processor-specifications and running processors, all connected together to accomplish some purpose.

To import a program into an OS instance, it is presented to the import-export processor in a standard distribution format called a distribution bundle. A distribution bundle has the processor-specifications, architecture specifications that state how to create and hook together processors in the program, data that the program uses, and install information such as adaptors, security requests, and meta-data to tag onto files.

After installation, a processor-specification is the same as any other data: it is held inside a persistent processor as data. A processor-specification is only distinguished from other kinds of data by meta-information. To create a processor, the creator processor is given the name of the processor holding the specification. The creator processor connects to the specification-holding processor, retrieves the specification data and uses it to create a new processor.

### 3.3.2 Specialization

There is a twist to distribution bundles and importing programs, however, which is the specialization process. This is the process by which a single source distribution is transformed into a machine-specific version that is optimized for particular hardware. The specialization may be performed in batch mode at some central distribution location, or it may be performed locally at the time the program is imported.

In effect, specialization performs the translation of the source code, so that portion of the import task is off-loaded from the Import-export processor.

Commercial programs will most likely be specialized in batch mode. The distribution media will come with an "ExecutableDistributor", that accepts a hardware description and gives back the specialization of the code that runs efficiently on that hardware. For example, a DVD-ROM may be loaded with a number of specializations, a database describing them, and an ExecutableDistributor. The distributor is run, given the hardware information, and it gives back a particular executable.

### 3.3.3 Importing data into an OS instance

The proposed OS interface uses the abstraction that everything is a processor, so a data file is, naturally, a processor. Importing a file into an OS Instance consists of translating the data-format via a pair of Import adaptors, and placing the result into a "holder" processor. Each holder processor contains only structured data. The data structures are associated with the Holder, and the imported data is used to create instances of those structures. Holder processors are persistent, and have meta-data attached, so they can be found by using the OS's search service.

To make a Holder processor, the OS abstraction implementation has a pre-defined specification of a file-processor. This is used to create the new, persistent, file-processor. At the time of creation, the meta-data of the file-processor is put into the name-discovery processor, and a pure name is generated that is put into the OS instance's internal name-space mechanism.

In the name-discovery processor, the name has attached meta-information supplied by the distribution bundle. 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 file-processor's pure name is placed into the name-space, 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 when a pure name is used to communicate to a processor. The OS instance internally uses the pure name to look up the hardware-specific information it needs (this activity is performed by the run-time system, which implements OS instance's internal name-space mechanism).

## 3.4 Creator processor

The creator processor is the means by which new processors are created. This is the analog to running a program in other OSs. The creator takes a processor-specification and creates a processor from it, then inserts the name of that new processor into some namespace.

The implementation of the creator processor is normally found in a run-time system. The run-time does some hardware-specific and native-OS specific thing to initialize and begin the run of code.

Creation is initiated by sending the name of a spec-containing processor to the creator, along with an indication of the namespace to insert the newly created processor into. The creator-processor makes a name for the new processor instance and registers it in the name-space, as well as sending the name plus meta-info to the name-discovery processor.

The creator processor can also destroy processors. 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 for the destruction operation owned by the targeted-for-destruction processor.

## 3.5 The User Interface

When a user sits at a physical display, the code that paints what the user sees is running natively on the local machine. Such code is called a Display, and is always native to the hardware the users is sitting at. The Display takes the place of a "shell". Each Display instance connects to a particular OS Instance. What the Display paints is a list of DisplayElements that are sent to it by a Visualizer. A Visualizer sits inside each processor that can be looked inside of. Hence, upon initial connection, the Display paints a visualization of the OS Instance processor that is sent to it by the Visualizer in the OS Instance. The Display may then browse among the processors that are inside the OS Instance. When a Display "goes into" a processor, it requests a connection to that processor, then receives a DisplayList from the Visualizer inside that processor.

When a Display attempts to connect to an OS Instance, the machine that Display is running on is registered as an external processor. This means the Display is a processor that is outside the OS Instance. Creating such a connection and the communication process is handled by the External Interface processor. This arrangement hides all hardware details from anything inside the OS Instance. Any application that includes things that a user can look at is written according to a standard protocol for Visualizers. This standard protocol might be translated during specialization for performance reasons, but this is hidden from the application.

An additional feature is that a processor may search for a Display that a particular user is looking at, or search for a Display that fits some other criteria, by using the name-discovery processor. Once it has found such a Display, it may initiate sending a visualization to it. This might be useful, for example, in distributed development environments. When one programmer modifies something that has a "watch" placed on it, the system can look to see if the owner of the code is sitting at a Display anywhere. It can then negotiate with the Display to pop up a message to that user about the code that is being modified.

## 3.6 Name Discovery Processor

The name discovery processor is used to search for the pure name 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 are specific to each OS instance. They must be searched for by a program. The result from a search sequence is the name of a processor that is then given to the name-space mechanism 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 name discovery processor is given data about each processor when that processor is created. Creation causes both the name of the newly created processor to be added to a name-space, and the name plus meta-data to be given to the name discovery processor.

The most common implementation of the name discovery processor will be using a relational database. For this reason, the search protocol has a resemblance to a SQL session.

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.

## 3.7 External Interface

The external interface allows communication with processors that are outside the OS instance. This is one of the main ways that the boundary around the OS Instance is enforced.

As an example of using the External Interface, a remote user wants to log in to the OS instance and run a program. They start a Display natively on the machine they're sitting at, then use it to connect to the external interface processor. The external interface initiates the authentication process, after which it allows communication. If the user is connecting via a Display, then the OS Instance's Visualizer creates a DisplayList. The DisplayList will only include the things that user is allowed to see. The user's authentication must pass the requirements for the Visualization operation on each processor Visualized (this limited vision improves security).

The external interface registers external processors, including instances of native OSs running on machines, and servers listening on ports of URLs. This registration creates a tunnel from the external processor into the OS instance's name-space. The registration creates a pure name for the external processor and places it into both the name discovery processor, and the name-space, the same as if the external processor were just created inside the OS Instance.

An important part of registering the external processor is specifying the data-structures that may be communicated to and from it, by registering adaptors. The adaptors parse incoming bit-streams into internal structured data, and vice versa. The parsing translates incoming primitive data types into standard interchange format (see section 4.2.1), and from there into internal format, and vice versa. Note that the parsing performed by the adaptors should not incurr a performance penalty in most cases because of the slowness of external communication compared to processing rates.

## 3.8 Authentication Processor

The authentication processor is the abstraction for the security model in the proposed OS abstraction. It uses a challenge model: an authentication must pass the requirements for perfoming a particular operation on a processor. This model was chosen for its flexibility.

### 3.8.1 Security Model

The security model involves three things in security decisions:

1. Authentication – black box that represents the authentication performed by a User
2. Requirements – black box that holds a map from authentication contents to yes/no
3. Operation-type – a set of operations that each have a Requirements attached

An authentication is created by a challenge sequence: a processor, such as a Display, that wishes to have an authentication attached to it is given a challenge and must return the correct response in order to receive the authentication. The contents of an authentication is hardware specific, it is a black box to all processors except the Authentication Processor. The choice among standard challenge sequences is also hardware specific. It may be a password on one system, and a finger print signed by a digital key embeded in a USB dongle on another system.

Authentications are used when one processor sends a communication to another processor. The sending processor attaches one or more authentications to its request. Meanwhile, the receiving processor has Requirements attached to each command it has. When the request arrives, the Requirements for the command sent is selected by the receiving processor. Then the authentications are fed into the Requirements one by one until the answer is either yes, or all authentications have been tried.

The implementation of testing whether the sent authentications pass the requirements for the requested operation is hidden. It is specific to the OS abstraction's implementation. All that is known is that the authentications get sent against the requirements, but the details of how are left unspecified.

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

### 3.8.2 Authentications, requirements and using the authentication processor

An authentication can be given by one processor to another. However, the received authentication is marked as a proxy. The proxy mark remains when the authentication is used. The implementation of the namespace keeps the actual authentications and performs the marking, so the mark cannot be tampered with. The sending processor decides which authentications to attached to the request it sends, but the namespace performs the actual attachment.

It is up to each receiving processor to decide how it wants to use proxy-authentications. It tells the challenge mechanism whether to accept proxies or not.

A performance shortcut allows the creator of a connection to attach a default authentication to it. After establishing the connection with a default, every subsequent communication automatically carries that default authentication (at least logically, if not physically). This can speed up challenges, because the ends of the connection cannot change, so the challenge only has to be performed once for each kind of operation requested.

Requirements are generated in an implementation-specific way. The details that go into a requirements are stored with each specification, and are first created when the specification is imported into the OS instance. After import, the details can be manipulated by authentications that pass the requirements for manipulating the requirements. The distribution bundle has a standard language for specifying how to generate requirements. However, the resulting requirements will be different on different systems, due to the differing security implementations in the underlying native OSs. At some future date a single, uniform mechanism will be decided upon.

A request to create a new processor is accompanied by the authentications to attach to the new processor. Any authentication that the requesting processor owns may be given to the created processor. Alternatively, the creator can be told to obtain a brand new authentication from the authentication processor. The new authentication would be the result of an authentication sequence engaged in by a specified source processor, such as a Display (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).

## 3.9 Name Spaces

A name-space is how processors communicate. The implementation of a name-space is the physical thing that gets information from one processor to another. The name of the destination processor is used to steer the information to it. In the proposed OS abstraction any communication mechanism, such as a network, is considered to implement a name-space.

Every processor has an internal namespace. Presence of a name in that namespace defines "inside the processoe" vs "outside the processor". The name of any processor known to the internal name-space is "inside the processor".

However, there also exist namespace tunnels. Such a tunnel creates a local name for a processor that lives in an external name-space. This makes the external processor look like it's in the namespace too. A real world example is what a router does to make a local area network look like all its machines are on the internet when in fact only the router is. This is what the external interface does, it creates a namespace tunnel for external processors.

The implementation of name-spaces is hardware specific. However, the use of pure names, that have associated black boxes, hides the implementation from applications. The info that is implementation specific is kept locked away inside black boxes, while the pure names which an applications receives from the OS Instance, are used the same way on all hardware. This gives a clean generic usage for applications while allowing arbitrary hardare details.

# 4 Enforcing Genericity

Now that the essence of the proposed OS abstraction has been seen, a little more detail on how the abstraction remains generic while accomodating a variety of hardware details is in order.

## 4.1 Data-format boundary

A data-boundary around an OS instance is one part of achieving genericity. To enforce such a boundary, hardware details embedded in data that crosses must 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, encryption, TCP/IP protocol, and so on)
- Instruction set (in 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)

## 4.2 Genericising an application's view of 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 abstraction in the following ways:

1. Communication format is abstracted. This is done 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 specialization process. All programs runnable within the proposed OS must be obtained from a specializer. The proposed OS abstraction provides no mechanism to run images pre-compiled for specific hardware, except when obtained by secure means from a trusted batch specializer. Most code will be specialized during the import process, which means that the executable form is generated by the OS Instance itself.

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. This hides all location information (which carries hardware implications).

Sometimes a programmer consciously 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 interface, at which time adaptors were specified that translate the program-provided array to the bit-stream the hardware requires. This keeps all hardware specific code in the adaptor, the external interface, and the OS instance's name-space implementation.

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 processor is registered with the OS instance, in addition to a name for that external processor being created and placed into both the name discovery processor, and the OS instance's 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.

### 4.2.1 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 that 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, 128-bit, or 256-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 abstraction defines protocol-adaptors that 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 wrapped in the IP packet protocol, then in the TCP connection protocol, then in a primitive data-type format (int, endian..), then inside the HTTP protocol, then inside the HTML protocol. For the web-server to understand the data itself, the data 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 that does 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 normal programs. They are distributed in the OS's distribution bundle format, so they need be written and compiled-from-source only once, and can then be used on every implementation of the OS interface.

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 be sent to an external processor, but this is deprecated in favor of registering the adaptor used in the cast with the external interface and having it automatically applied.

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.

# 5 Results

We have not yet implemented the proposed OS abstraction. It is underway as part of the EQN-Lang project on sourceforge, and as the CTOS project, also on sourceforge.

# 6 Conclusion

This paper has described an OS interface that should be easy to implement on top of existing OSes and provides a generic interface for application programs to use. 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, and even primitive data format.

This genericity is accomplished by the use of an automated specialization process, the use of the processor abstraction for communication, the inclusion of built-in processors for OS services, and the use of adaptors to enforce a data-format boundary.

# Bibliography