A Proposal for Computer System Schemas

Brent B. Welch
Xerox PARC
3333 Coyote Hill Road
Palo Alto CA 94304

Abstract

This position paper proposes the use of an object-oriented database to model the high-level aspects of distributed systems. Components of the system that are modeled by objects are applications, services, libraries, files, IPC channels, hosts, peripherals and users. In this approach a database schema defines the possible interactions among the objects in the system, and the contents of the database model the current state of the system. Queries on the database can be used to answer questions about the system, to construct communication paths between objects, and to choose the appropriate location and medium for the display of information. There are also general applications to system administration, maintenance, performance tuning, and debugging.

1. Introduction

One of the big problems with today's computer systems is that they have been built up over time into an ad-hoc collection of programs, data files, and hardware. Even a brand new workstation with a fresh OS installation is opaque to the new user and perhaps even the system administrator. What files are safe to remove? Why does a particular application work for one user while it fails for another? The source of these problems is that there is no formal description of the relationships among the components of the computer system. Instead, the system has been handcrafted by a team of experts so it requires an expert to modify it.

This paper proposes using an object-oriented database that records the interdependencies among the components of a distributed computing system as well as the current status of the objects. I will use the term "computer system schema" instead of an "object-oriented database", however, to emphasize that the actual things being modeled by the database are regular files, hardware devices, running programs, and people, as well as data objects in a database. The information in the database defines things like the requirements of applications, user preferences, host configurations, and service attributes. The information in the database can be used for system administration and for trouble shooting, both of which are becoming more difficult as systems scale up into more complex distributed systems. With this in mind, the term "object" in this paper will refer to a high-level system component like an application, a library, a file, a host, a printer, or a user.

The computer system schema provides a formal description of the status and interactions among the components of the computer system. In this respect the schema provides the same kind of structure as the class hierarchy of a system like Smalltalk. The disadvantage of language-oriented systems like Smalltalk, however, is that all the objects are usually defined within an application or within a set of applications that share a single address space. The structure of objects within the system is well defined, but as some point the boundary of the Smalltalk world is reached. There is a need for a higher-level structure that supports the interaction of more independent sets of applications.

Current UNIX systems, in contrast, are composed of a collection of programs and data kept in their file systems. The execution environment of a UNIX process adds more components such as environment variables, open I/O descriptors, and signals. Currently there is no mechanism to formally define how a UNIX application interacts with its surrounding environment. The file system, for example, can be viewed as a massive collection of global variables. There are no interfaces that define what files (i.e., variables) should be used by which programs (i.e., procedures). The proposal made in this paper is for a higher-level schema that models the interactions among diverse components. The goal is not to overly constrain what is possible. The goal is to provide some structure so that it is easy to determine what the right thing to do is and what the current state of the system is.

The benefits of the schema-based approach are two-fold. First, computer systems ought to able to manage themselves to a much larger degree than is done today (especially in the UNIX world). A machine-readable schema and database will make it possible for the objects of the system to analyze their surrounding computing environment and react to changes in the environment. Second, explicitly recording dependency information will make it easier to reuse existing software components. Currently the way applications work together is either not well defined or it is hardwired into the applications, preventing a more general use. Without a schema-based infrastructure I assert that we will fail to scale up our distributed systems and keep them well integrated. Instead, we will end up with an unrelated collection of applications that are difficult to manage and clumsy to use. In short, we will remain stuck in our current state of affairs.

2. What is a Computer System Schema?

Let us abstractly define a computer system schema as a set of typed relationships among typed objects. Objects have types, UIDs, slots, and methods. Each slot is an attribute-value pair, and these are the basis for queries about the system. The type of an object (its class) is a label for the collection of slots and methods that it supports. One could use inheritance to define a class hierarchy, although initially a simpler system without inheritance will still be quite useful. With this simpler view, one can imagine the database as a set of tables with formats defined by a schema and contents that model the state of the system.

There are five aspects of the computer system schema: method information, dependency information, status information, parameter information, and events triggered by changes in the database.

The methods of an object define what can be done with the object. Methods include administrative ones like "make", "install", and "help", as well as object-specific methods. A mail system might provide "read-mail" and "deliver-mail" methods, for example. These methods are high-level methods that roughly correspond to executing a process or a set of processes in the UNIX environment. This is somewhat different than the method suites that correspond to RPC interfaces exported by services. While it is possible to store the RPC interface definition in the database, it is more important to record what servers export what interfaces and what clients import them. Thus the methods of our objects correspond to high-level system interactions, and the more detailed, language-specific interactions are not necessarily modeled by the object system proposed here.

It is important to record dependencies among objects so that tools can reason about what is possible, what it takes to add a new object (i.e., application) to the system, and how an object can be used by other objects. It is important to record status of objects so that tools can determine the best way to achieve a desired result, why an operation failed, and how well the system is performing. If the state of the system is modeled well enough in the database, then many problems can be resolved by queries such as "find the display nearest Brent", "what servers does this workstation depend on", or "what programs operate on objects of this type".

Applications can be parameterized by the contents of the database. There will be a relatively large static or semi-static part of the database that captures the use of files and other resources (machines? printers? gateways?) by applications. The name of a resource can be changed in the database without recompiling the program. Similar functionality is achieved in the Machintosh OS by breaking files up into resource and data forks. Resources can be edited after an application has been compiled, so things like file names and even the text in dialog boxes can be easily customized. In UNIX, environment variables are used to achieve this functionality. This is done in an ad-hoc manner, however, plus there is no way to tell without reading source code which programs use which environment variables. A more disciplined approach is needed.

In order to simplify the specification of an object, we limit the resources named in the database to those that are always needed by the object for normal operation, and we exclude resources that are functions of the object's inputs. The C compiler, for example, always uses the standard C library, but it will read a different set of source files and include files depending on its command line arguments. It would be tricky and expensive to try and capture the input-derived dependencies in the schema. However, I would like to capture the fact that my mx editor reads /import/lib/mx/default.mx and (optionally) ./.mx and ~/.mx each time it is run. Similarly, many applications depend on particular versions of dynamic libraries.

Other parts of the database will be dynamic, especially status and location information. Changes to the database should cause events that can be analyzed by other objects of the system. This reflective capability enables triggers such as file-modified callbacks, new mail notification, tracing and logging for debugging, graceful error recovery, and a general source of signals for analysis. We can express callbacks as a kind of query on the future state of the database. It might also be possible to do "time shifting" and make queries as if it were some other time than the present.

Applications will have to change a bit in order to gain the full benefits of a schema. First, they should know "who they are" (i.e., what object represents them in the database) so they can indirect through the model. Instead of compiling a file name into a program, the program should query the database to get the name of the file. This is in the spirit of Mac resource forks. Secondly, applications should update the schema with status information. This is the means by which applications project themselves onto the rest of the computer system. Other applications can query the database to determine if the application is ready to provide service, is in need of a debugger, or perhaps what its level of activity is for the purposes of load balancing.

3. Related Work

There are a number of standard proposals and prototype systems that include features described in this paper. In some approaches, a completely new environment is specified. One example is PCTE [PCTE90], the Portable Common Tool Environment. This standard specifies an system where objects have attributes and links among objects, and the objects in the system are maintained in a database. The specification includes many features such as replication, transactions, auditing, accounting, and access controls. The Emaraude project is a partial implementation of PCTE. The PenPoint system designed for portable computers is another excellent example[Carr91]. Applications are formally installed into the system, dynamically linking themselves into the system's class libraries.

On another front, the Object Management Group (OMG) is currently in the process of selecting a reference specification for an Object Request Broker (ORB) [Dyson91]. The ORB is envisioned as the glue between distributed object systems that allows invocation of remote object methods. In a sense, it is a manager of object managers. It fulfills the role of binding agent among distributed objects, which is a crucial function to standardize if object oriented systems are to be able to interoperate.

The area of network management has received considerable attention. The OSI Management Information Base (MIB) [Labarre91], for example, defines an object-oriented model for dealing with hosts, gateways, and routing table information. A number of network management systems have been proposed [Klerer88][Shim90], which reflects the need for more sophisticated tools to manage todays networks.

The X-station is a system that is used to dynamically compose tools into more complex tools. The key aspect of the system is that each tool is first split into two halves, a functional part and a user interface part that interact with a well defined message protocol. This allows the tools to be coordinated by a tool manager that is driven by scripts associated with each tool. The system has been used effectively to coordinate the use of softare tools for biologists[Nanard90].

So, it appears that object-oriented techonolgy is being advocated to improve systems in various ways. However, systems are either "new world" approaches that require building new tools, or they are targeted for a narrow topic such as network management. There is still a need to apply object-oriented approaches to managing the day-to-day aspects of computer systems, especially UNIX.

4. Example Uses of the Schema

This section gives a few examples of how a computer system schema would be useful in todays systems.

4.1. Client-Server Binding

An important application for the computer system schema is to use it as a binding agent for clients and their servers. Servers register themselves in the database with various attributes, and clients make queries in order to find the best server for their current needs. This is no longer the semi-static registries of, say, the Amoeba directory service that maps names to capabilities (i.e., server handles), but instead a registry of relations and attributes that a client can query and reason about. The result of this style of client-server binding is that all the gory details of communication protocol connection setup can be hidden behind a library interface. Clients should know only that they need a server with particular, high-level attributes (like location, load, and availability), and not worry about low level attributes like network addresses and protocol suites.

Once system objects indirect through the schema to set up bindings, then the database implementation can log the current state of the system - who is bound to who. This will be a great help in understanding, debugging and optimizing the system. For example, suppose a client gets a failure during some operation on a server. The database should record that the server is executing on a particular machine and it is a client of a number of other servers. A debugging tool should be able to trace through this information and figure out something like "the machine running the server is down" or "the file server the server uses is down".

4.2. Host and Server Monitoring

Host and server monitoring are obviously important if we are to have reliable systems. A network/host/server monitor can periodically query objects with null RPCS or ICMP echo probes. In this example, there is an external source of status information (the host/server monitor), and its updates to the database trigger callbacks to other interested parties. With ISDN coming soon, it should be possible ring the computer operator's phone and play a canned message when a server machine crashes. A server-server will also be a client of this interface, restarting failed servers and logging these events.

4.3. UNIX administration

Let's begin with the bootstrap process of a UNIX workstation. The kernel spawns the first user-level process, /etc/init. Init forks a shell that interprets a script of things to do during bootstrap. This script has grown into a collection of scripts that contain a hodge-podge of programming fragments that initialize various software objects. Instead, one could take a table-driven approach that queried the database to find out what objects need to be initialized, and then invoked an initialize method for the objects. Dependency information in the database could determine the order that things need to be done, including what could be done in parallel. Each initialization method would terminate by updating the database with status information so the capabilities of the host could be known to the rest of the network.

To ensure robustness, the database entry for each host should be cached in its private file system so the machine can boot up as much as possible in the absence of supporting servers. After initialization the host could see if its cached configuration information is up-to-date. If not, it could selectively reinitialize itself, or simply reboot after updating its cache.

The benefits of configuring hosts in this way are two-fold. First, it will force the vendors and the system administrators to specify what the requirements of each software subsystem are and therefore make it easier and less error prone to configure a particular machine. Secondly, the state of the machine will be advertised so it will be possible to determine what the capabilities of the machine are (i.e., what services it is running).

The initialization of a user's login environment can be handled similarly. Currently a script, .login, is run in order to set up the environment. The .login file is a program that configures a user account for the use of various software packages and that customizes the account to match the preferences of the user. A program is arguably the most flexible approach to configuration and customization, but it is also risky and mysterious to the uninitiated. It would be better to confine the programmable aspects to the domain of experts, and only provide organizational tools to the end-user. In a schema-based system the .login file could be replaced with a database entry that defines user preferences, including what software packages will be used, and another set of database entries that define what the requirements of the software packages are. A "join" of these relations would produce a set of actions to take that would configure the user's environment for using the software.

5. Implementation

This section outlines an implementation approach. Currently this still being designed, so there is not yet even a prototype implementation. It is potentially a vast amount of work to create a schema for all the UNIX tools in use. However, some of the work could be automated (i.e., extraction of library dependencies), and other important tools could have schema descriptions created by hand.

While a schema is conceptually a centralized database, in reality it will have to be distributed among the hosts and among the software components themselves. Something akin to a UNIX Makefile, call it a "Schemafile", can be used to describe a system object. The full schema is conceptually a union of all the Schemafiles in the system. A Schemafile would list the attributes and methods applicable to the object. A Make method, for example, would describe how to invoke a build tool (i.e., regular make) and perhaps what the possible build targets are (i.e., clean, install, testsuite). A Verify method would check the components installation to make sure it is complete. Then, of course, there will be one or more methods that actually execute the tool being described. The Schemafile also specifies external dependencies such as use of files, environment variables, and inter-process interfaces. The later can be RPC interfaces, the X protocol interface, UNIX standard I/O, UNIX sockets, and other mechanisms.

At the top level there needs to be an organization of the objects and their associated schemafiles. The UNIX file system can be organized in a canonical way to achieve this. Currently here at PARC, each software package is supposed to have a "top" directory that has a few standard scripts. One script is used to "enable" the environment for use of the application, and another script is used, to "bringover" default versions of startup files. The top directory also contains a README file that gives a short description of the application. This simple arrangement could be extended so that each application has a Schemafile and a larger set of methods.

A per-host Schemafile could be used to describe the configuration of the host, including the set of processes that are created during bootstrap. The per-host Schemafile would invoke the "initialize" method for the set of applications and daemons it needs. These methods would be found by consulting the Schemafiles for the individual applications specified in the per-host Schemafile.

A per-user Schemafile could be used to specify the applications to be used and other personal preferences. This replaces the functionality of the user's .login and .cshrc customization files. If inheritance is supported by the Schemafile interpreter, then user classes can be defined and maintained by the system administrator.

6. Conclusion

There are five important aspects of a computer system schema.

(1) Method information describes what high-level systems operations can be applied to the objects in the system.

(2) Dependency information among objects is recorded so we can know what is possible and how to achieve it.

(3) Parameter information is recorded so that tools do not have dependencies on the computer environment hardwired in.

(4) Status information is recorded so we can know why something is not possible and how to correct the problem.

(5) Changes to information generate callbacks so that tools can react to a dynamic environment.

Many of the ideas contained here are perhaps just new ways of viewing mechanisms that have been around in computer systems for some time. My main goal is to focus attention on the overall organization and architecture of the system. Systems today limp along without an overarching schema, but they could be much better integrated if they had one. New tools, in particular, should be built from the beginning with an integrated schema that defines the system integration. Applications should rely on a fairly high-level interface to the rest of the objects of the system so that the implementation details are hidden (e.g., Sun RPC vs. TCP/IP). The implementation of the schema and the associated communication infrastructure can gradually improve as we learn more about what it needs to provide. The key thing is that applications should have access to a model of the surrounding system and a well defined way of learning about changes to the surrounding system. The system has to be cognizant of change so it can behave reasonably. Before it can understand change, it has to have a basic schema upon which to reflect.

References

Carr91. R. Carr and D. Shafer, *The Power of Penpoint*, Addison-Wesley Publishing, Feb. 1991.

Dyson91. E. Dyson, ''Domain of Objects: The Object Request Broker Request'', *Release 1.0*, Mar. 1991.

PCTE90. ''Portable Common Tool Enviornment (PCTE) Abstract Specification'', Standard ECMA-149, European Computer Manufacturers Association, Dec. 1990.

Klerer88. S. M. Klerer, ''The OSI Management Architecutre: an Overview'', *IEEE Network 2*, 2 (Mar. 1988), 20-29.

Labarre91. L. Labarre, editor. ''OSI Internet Management: Management Information Base'', Network Working Group RFC 1214, Apr. 1991.

Nanard90. M. Nanard, J. Nanard and P. Pingand, ''An Object-Oriented Kernel for Dynamical Software Integration'', *Proc. First International Conference on Systems Integration*, Apr. 1990, 584-591.

Shim90. Y. C. Shim and C. V. Ramamoorthy, ''Monitoring and Control of Distriubted Systems'', *Proc. First International Conference on Systems Integration*, Apr. 1990, 672-681.