Journal of Advances in Science and Technology
Vol. III, No. IV, February-2012, ISSN 2230-9659

Virtual File System Based Embedded Software Development: Underlying Technology

Parveen Kumar

Research Scholar, CMJ University, Shillong, India

--♦------------------------------------

INTRODUCTION

The breadth of technologies required to bring performance benefits to our customers is remarkable. Maintaining up-to date knowledge in a number of rapidly changing fields as wide apart as low power applications to software generation and verification, from advanced signal processing to FPGA technologies is a great challenge for our global research and development teams.

With the ever increasing penetration of embedded systems in society and the related increase in investments by industry to develop such systems, the investments in embedded software engineering technologies increases as well. Forecasts [2] predict continuing increase in embedded systems specific technologies. This section uses the term 'technology* for any method, technique, process or tool that can be used to support a certain development activity.

The market for software engineering technologies is largely fragmented. There is no clear market leader, and there is no supplier present that fully supports the whole development chain of embedded products. There are different suppliers for requirements engineering technologies, different vendors for design technologies, etc. Most dominant is the sales of software tools. Tools imply to support or be supported with a methodology. Some suppliers provide methodologies with their tools, while others support generic methodologies, such as UML (OMG's Unified Modeling Language) or MOF (MOG's Meta Object Facility).

As the market of technologies is fragmented, so are the technologies themselves. Technologies in general are stand-alone solutions for specific problems. As embedded software development is collection of complex and technical problems, several technologies are applied in parallel [3]. It does not take very much time or experience to observe that this lack of integration is a cause for problems too. Technologies are used separately but depend on each other, interfaces are not defined, inconsistencies occur, etc. Despite this loss of quality due to lack of integration, there is moreover a loss of time, effort and money due to duplication, redundancy and cost of non-quality. As time pressure is prominent in many market domains where embedded systems are sold, this indicates a potential gain in time-to-market. Time-to-market gains in embedded systems development cause high potential revenue gains due to earlier market introduction and therefore deeper market penetration. Industrial companies therefore support initiatives for time-to-market reduction strongly.

Though many software product and process technologies are already available, the embedded software domain puts specific demands to the application of these technologies. Dedicated research results and products are present for software architecture development and assessment, requirements engineering and validation, software process improvement, and tools to support all these technologies. However the major disadvantages of these technologies are that they do not take into account the specific needs for embedded systems and that they are applied "stand alone'", which in many cases is not very effective and leads to disappointing results.

The embedded systems industry puts specific demands to the usage of such methodologies, such as the large dependency on real-time features, limited memory storage, large impact of hardware platform technology and the related cost drivers of the hardware, etc. The existing software engineering methodologies do not distinguish the specific impacts or necessary customisation for the embedded domain, nor is it indicated how they should be used specifically for each specific area within this domain, i.e. automotive, telecom, consumer electronics, safety critical, etc. The embedded software domain puts dedicated pressure on these methodologies. Reasons for this are the high complexity of these products and the dependency in this domain 011 innovative highly technical solutions.

Furthermore, the embedded domain is much more driven by reliability, cost and time-to-market demands. This makes the embedded domain a specific area for which available generic methodologies need to be adapted.

Advances in process technology and the availability of new design tools are broadening the scope of embedded systems; from being implemented as a set of chips on a board, to a set of modules in an integrated circuit. System-on-Chip (SoC) technology is now being deployed in industrial automation, enabling the creation of complex field-area intelligent devices. This trend is accompanied by the adoption of platform-based design, which facilitates the design and verification of complex SoC through the extensive re-use of hardware and software IP (intellectual property). A further important aspect of the evolution of embedded systems is the trend towards networking of embedded nodes using specialized network technologies, frequently referred to as Networked Embedded Systems (NES).

System-on-Chip (SoC) represents a revolution in integrated circuit (IC) design, enabled by advances in pro- cess technology, which allow the integration of the main components and subsystems of an electronic product onto a single chip or integrated chipset [1]. This development has been embraced by designers of complex chips because it permits the highest possible level of integration, resulting in increased performance, reduced power consumption, and advantages in terms of cost and size. These are very important factors in the design process, and the use of SoC is arguably one of the key decisions in developing real-time embedded systems.

SoC can be defined as a complex integrated circuit, or integrated chipset, that combines the main functional ele- ments or subsystems of a complete end product in a single entity. Nowadays, the most challenging SoC de signs include at least one programmable processor, and very often a combination of at least one RISC (reduced instruction set computing) control processor and one digital signal processor (DSP). They also include on-chip com- munications structures - processor bus(es), peripheral bus(es) and sometimes a high-speed system bus. A hier- archy of on-chip memory units, as well as links to off-chip memory, is especially important for SoC processors. For most signal-processing applications, some degree of hardware- based accelerating functional unit is provided, offering higher performance and lower energy consumption. For interfacing to the external world, SoC design includes a number of peripheral processing blocks consisting of analogue components as well as digital interfaces (for example, to system buses at board or backplane level). Future SoC may incorporate MEMS-based (microelectro-mechanical system) sen sors and actuators, or chemical processing (lab-on-a-chip) D.

All interesting SoC designs comprise both hardware and software components. These include programmable processors, real-time operating systems, and other elements of hardware- dependent software. Thus, the design and use of SoCs not only concerns hardware - it also involves system- level design and engineering, hard- ware—software tradeoffs and partitioning, and software architecture, design and implementation.

System-on-a-Prog rammable-Chip Recently, the scope of SoC has broadened. From implementations using custom IC, application specific IC (ASIC) or application-specific standard part (ASSP), the approach now includes the design and use of complex reconfigurable logic parts with embedded processors. In addition other application-oriented blocks of intellectual property, such as processors, memories, or special purpose functions bought from third parties are incorporated into unique designs.

These complex FPGAs (Field-Program- mable Gate Arrays) are offered by several vendors, including Xilinx (Virtex-II PRO Platform FPGA, Virtex-IV) and Altera (SOPC). The guiding principle behind this approach to SoC is to combine large amounts of reconfigurable logic with embedded RISC processors, in order to enable highly flexible and tailorable combinations of hardware and software processing to be applied to a design problem. Algorithms that contain significant amounts of control logic, plus large quantities of dataflow processing, can be partitioned into the control RISC processor with reconfigu- rable logic for hardware acceleration. Although the resulting combination does not offer the highest performance, lowest energy consumption, or lowest cost - in comparison with custom IC or ASIC/ASSP implementations of the same functionality - it does offer enormous flexibility in modifying the design in the field, and avoids expensive Non- Recurring Engineering (NRE) costs associated with field changes. Thus, new applications, interfaces and improved algorithms can be downloaded to products already working in the field.

Products in this area also include other processing and interface cores: these consist of multiply—accumulate (MAC) blocks aimed at DSP-type dataflow signal- and image processing applications, and high-speed serial interfaces for wired communications such as SERDES (serializer/'de-serializ- er) blocks. In this sense, system-on-a- programmable-chip SoCs are not exactly application-specific, but not completely generic either.

Another important facet of the evolution of embedded systems is the emergence of distributed embedded systems, frequently termed networked embedded systems, where the word "networked" signifies the importance of the networking infrastructure and communication protocol. A networked embedded system is a collection of spatially and functionally distributed embedded nodes, interconnected by means of wireline and/or wireless communication infrastructure and protocols, and interacting with the environment (via sensor/actuator elements) and each other. Within the system, a master node can also be included to coordinate computing and communication, in order to achieve specific objectives.

Controllers embedded in nodes or field devices, such as sensors and actuators typically provide on-chip signal conversion, data and signal processing, and communication functions. The ever-increasing functionality, processing and communication capabilities of controllers have been instrumental in the emergence of a widespread trend for the networking of field devices around specialized networks, frequently referred to as field area networks. (A field area network is normally a digital, two-way, multi-drop communication link [61.) In general, the benefits of using specialized (field area) networks are numerous and include: increased flexibility through combining embedded hardware and software; improved system performance; and ease of system installation. upgrade, and maintenance.

ALTERNATIVES FOR FILE SYSTEM IMPLEMENTATION

Traditionally, file systems contain directory structures that are tightly bound to a particular file system implementation. These structures may be embedded, both logically and physically, in the file system and contain data that are specific to the file system implementation. Changing the directory structure of a file system can be extremely tedious: the file system code must be changed and rebuilt, new file system initialization code (mkfs) is needed, and new recovery code (fsck) is also likely to be necessary. Several areas of file system research could benefit from a generic directory structure that is implemented above the physical file system layer, allowing experimentation with directory contents and possibly alternative naming schemes.

Ail existing implementation of directory files was completed 011 Linux 2.2.14 and subsequently on Linux 2.4.2. This implementation consists of modifications to the Linux kernel NFS server code, but works with any NFS client. A directory-file -based file system is created by initializing a root directory file, called ROOT. The directory containing this file is then exported. The ROOT file is initialized with entries for and that both refer to ROOT. A11 entry in a directory file consists of:

- The user's name for the object (e.g., passwd)
- The system's name for the object in the underlying physical file system (e.g., f ile . 001; see below)
- The type of the object (e.g., directory or file)
- Any other information that may be necessary (e.g., server where the object is located)

For this implementation, file system objects are created in a flat namespace in the underlying physical file system(s). and are given unique object identifiers (a name in the namespace of the underlying file system). Thus, the directory entry associates the users's name for a file with a unique object id used by the system to retrieve object contents.

In the NFS server code, an exported file system is flagged as a "directory-file file system" if it contains a ROOT directory file (this will change in a future implementation, and is just a temporary hack). All directory operations for this file system are intercepted and interpreted in the directory file context. For example, if a readdir request is received, the corresponding directory file is opened, its contents are read, and the appropriate readdir response is constructed and sent to the client. Reading and Birch is implemented as a Cocoa application that creates a user-space NFS system, and NFS RPC calls (which only come over a local socket instead of over the network) are interpreted to render metadata search queries as files and directories. A simple Cocoa user interface manages the search queries the file system renders. Queries are composed using the NSPredicate and NSMetadataQuery classes, which access the Spotlight metadata database.

The basic object is the system is the Query, which contains three fields:

- The predicate. which is a Spotlight query stored in an NSPredicate object. An example query is "kMD It em Authors == '*Brian Eno*'^{M}.
- A boolean attribute isLeaf. "Leaf" queries show their results in the file system view; non-leaf queries do not. The purpose of non-leaf queries is to store a partial query, which can be composed with other queries. Since the partial query may match too many files, the non-leaf attribute prevents large, possibly uninteresting, result sets from being displayed.

- A set of names of other sets, called subordinates. Subordinate sets are always shown in the file system, and provide a way to save metadata search paths. For example, one could store a query, like "kMD Item Album == ' *Music for Airports*' ", inside a query that it is commonly used with, such as "kMD Item Authors == '*Brian Eno*'". There is no hierarchy implied here; two queries may contain one another in their subordinate sets, and conjunctions can be formed by specifying either one first.

In order to simplify the implementation of the NFS server, no file I/O requests are handled by the NFS server itself. Instead, files referenced in a query always appear as symbolic links, where the content of the link is the full path to the file elsewhere on the file system. This mechanism greatly simplifies the objects the Birch file system needs to keep track of: everything in the file system is either a directory,and thus a predicate, or it is a link to a file matching a predicate. We think this kind of file system — which only uses links to reference resources — is a very useful implementation strategy, since it simplifies and optimizes file access, once the file is looked up. This makes the file system analogous to other forms of search results, like Internet search engines, which produce as output not the content found, but merely links to the content. It is also much easier to properly implement POSIX semantics if the capabilities of the file system are restricted, which was invaluable given the short, development time of this project.

A direct approach to implementing file abstractions for hardware is through dedicated operating system resident file systems providing the abstraction. These file systems can be mounted at required locations within the operating system namespace and be used to control and access hardware through conventional file operations. Many popular operating systems alleviate the difficulty in developing file systems by defining standard, generic interfaces that file systems can plug themselves into. Examples include VFS [102] in Linux, SunOS VFS [74] and Installable File System [93] in Microsoft Windows. While file systems have to comply with these interface standards set by the operating system, the implementation logic is largely the file systems’ prerogative. Thus synthetic file system behavior could be incorporated by completing various file operations through necessary interactions with the hardware being abstracted. A more restrictive means of implementing kernel-based file abstractions for hardware in Unix-like operating systems is through devfs [5], which enables association of a single file in a global devfs directory (typically /dev) with a hardware device. Handlers for the various file operations are defined in the hardware’s device driver and registered with devfs.

Techniques exist to implement filesystems in user space and integrate them within the overlying operating system namespace. The general idea of shifting functionality from within the kernel to user space has been aggressively adopted by the GNU Hurd [64] and Exokernel [46] projects among others to several core aspects of operating system operation including inter-process communication, file systems, signal handling and networking. Such systems have a minimal microkernel at their core, while much of the operating system functionality is transferred out to user space daemons.

THE PLAN 9 MODEL

Plan 9 is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. Because commands, libraries, and system calls are similar to those of the Unix operating system, it is possible to port many Unix programs to Plan 9 with little or no changes. A casual user would find little difference between the two systems.

What distinguishes Plan 9 is its organization. The goals of this organization were to reduce administration and to promote resource sharing. Our programming style was minimalism. We believe that a small number of well-chosen abstractions can, with much less code, provide most of the function of a larger system. This is the approach that made the Unix operating system so much smaller than its contemporaries such as Multics. In building Plan 9, we generalized proven ideas from the Unix operating system rather than add new untried concepts.

Plan 9 is divided along lines of service function. Diskless CPU servers concentrate computing power into large multiprocessors: file servers provide repositories for storage: and terminals give each user of the system a dedicated computer with bitmap screen and mouse on which to run a window system. The sharing of computing and file storage services provides a sense of community for a group of programmers, amortizes costs, and centralizes and hence simplifies management and administration.

Since both CPU servers and terminals use the same kernel, users may choose whether to run programs locally on their terminals or remotely on CPU servers. Plan 9 provides this flexibility without constraining the choice. Therefore, both users and administrators can configure their environment to be as distributed or centralized as they wish. At work, users tend to use their terminals more like workstations running all interactive programs locally and reservmg the CPU servers for data or compute intensive jobs such as compiling and computing chess end games.

At home, connected via a dedicated 9600 baud line to work, users choose what they run locally and remotely to reduce communication cost. Some applications, such as the editor [Pik87], are split into multiple programs to make this choice even more flexible.

Figure in any Plan 9 section shows how we have configured our environment. Multiprocessor CPU and file servers are clustered in a few computer rooms and connected via 7 megabyte/sec point-to-point links [Pre88]. This permits the CPU servers to be used as high performance compute engines without becoming starved for data. Terminals are connected to the servers via lower speed, lower cost distribution networks such as the 10 megabit Ethernet [MetSO] and 2 megabit Incon [Kal. Res], By emphasizing the shared service clusters we can quickly and cheaply incorporate new technologies as they arise. At the same time, users wishing more autonomy can incorporate as much computing power as they wish in their own offices without losing the advantage of transparently sharing other resources.

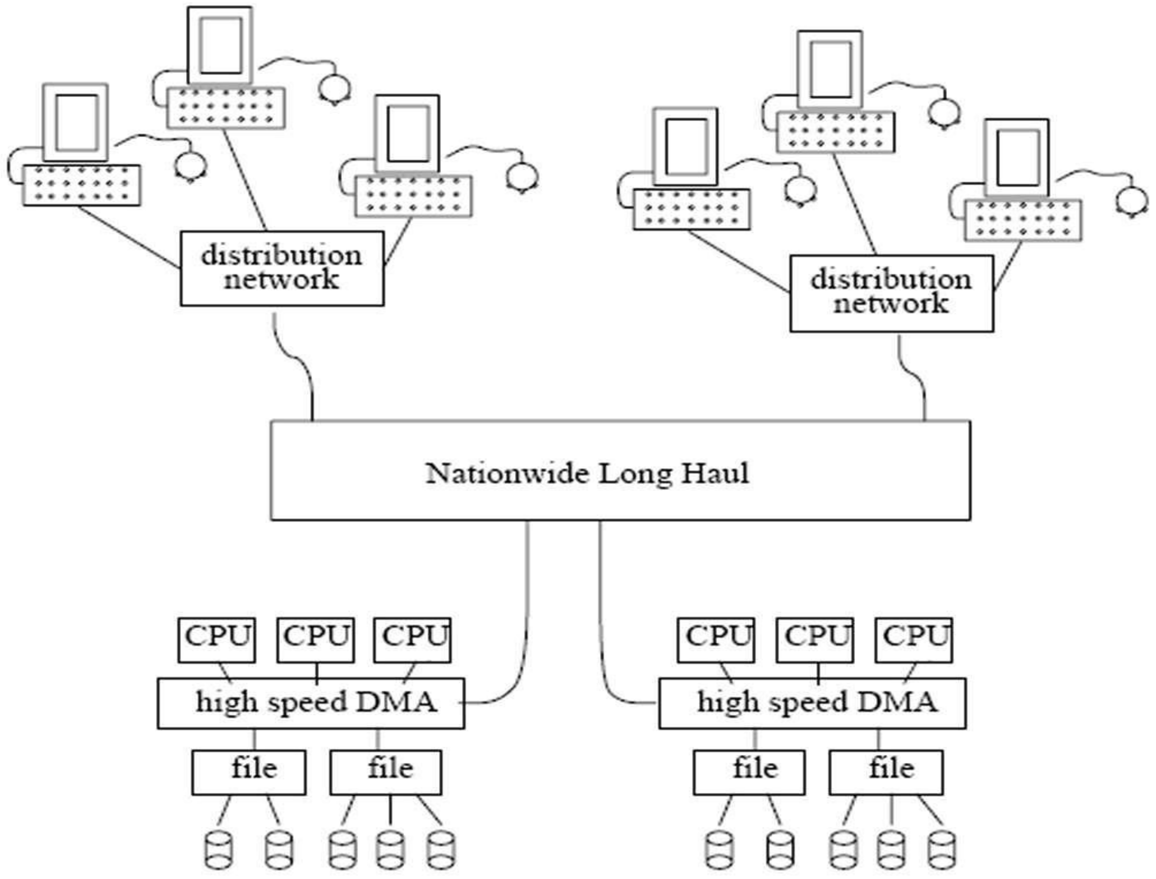

Figure - Plan 9 Topology

The rest of this section describes the features of Plan 9 that make possible such a flexible topology. For more information on hardware and use of the system, see our previous section [Pik90] . For details of the file server, see [Qui] .

All Plan 9 components are connected using this file system protocol. The code used to encapsulate the primitives into request and reply messages is 580 lines long. The mount driver is 899 lines long. Compared to the equivalent NFS code implementing vnodes and XDR this is tiny.

Iii Plan 9. every network interface is a file system. A gateway is a file server that serves its own network interfaces to other machines. A process that wants to get at a remote network connects to the gateway and mounts the gateway's interface to the remote network into its name space. Whenever the process accesses the interface, the mount driver will send the request to the gateway. Thus, the gateway sees exactly what the process does.

Plan 9 survives without local disk file systems thanks partially to hardware and partially to caching. The CPU servers do so because their links to the file servers transfer at a substantial percentage of memory speed. The file servers maintain large main memory caches for their disk file systems. These servers are configured with 128 megabytes or more of mam memory to ensure that there is plenty of room for cache. Getting a file from a file server is generally faster than it would be to get it from a local disk.

The view of the system is built upon three principles. First, resources are named and accessed like files in a hierarchical file system. Second, there is a standard proto-col, called 9P, for accessing these resources. Third, the disjoint hierarchies provided by different services are joined together into a single private hierarchical file name space. The unusual properties of Plan 9 stem from the consistent, aggressive application of these principles.

A large Plan 9 installation has a number of computers networked together, each providing a particular class of service. Shared multiprocessor servers provide computing cycles; other large machines offer file storage. These machines are located in an air-conditioned machine room and are connected by high-performance networks. Lower bandwidth networks such as Ethernet or ISDN connect these servers to office and home-resident workstations or PCs, called terminals in Plan 9 terminology. Figure shows the arrangement.

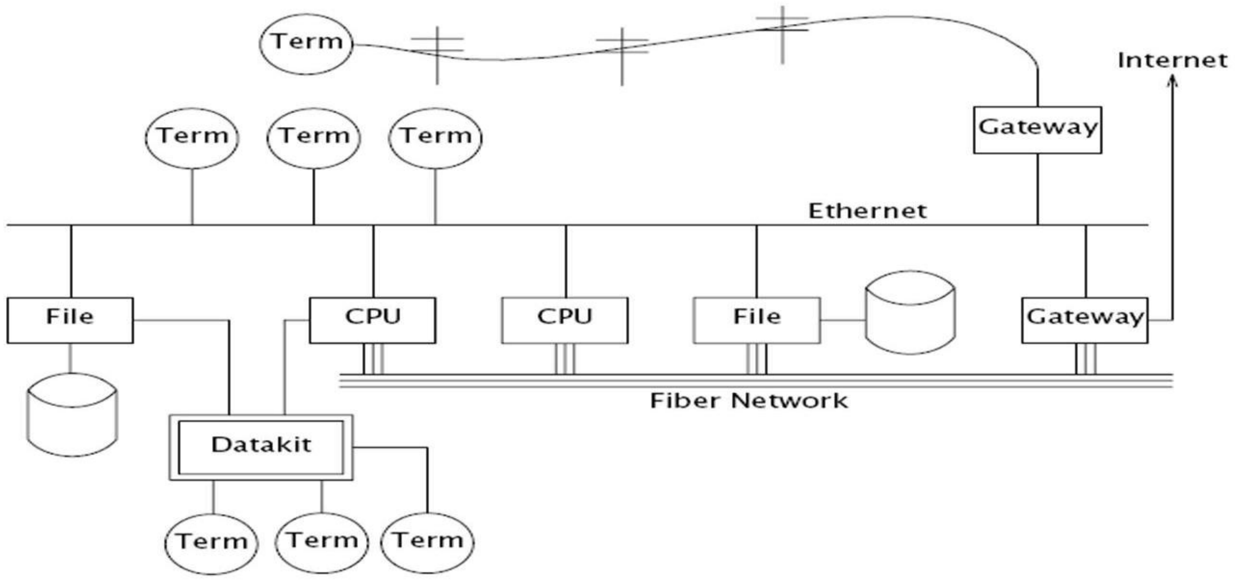

Figure. Structure of a large Plan 9 installation. CPU servers and file servers share fast local-area networks, while terminals use slower wider-area networks such as Ethernet, Datakit, or telephone lines to connect to them. Gateway machines, which are just CPU servers connected to multiple networks, allow machines on one network to see another.

The modern style of computing offers each user a dedicated workstation or PC. Plan 9's approach is different. The various machines with screens, keyboards, and mice all provide access to the resources of the network, so they are functionally equivalent, in the manner of the terminals attached to old timesharing systems. When someone uses the system, though, the terminal is temporarily personalized by that user. Instead of customizing the hardware, Plan 9 offers the ability to customize one's view of the system provided by the software. That customization is accomplished by giving local, personal names for the publicly visible resources in the network. Plan 9 provides the mechanism to assemble a personal view of the public space with local names for globally accessible resources. Since the most important resources of the network are files, the model of that view is file-oriented.

The client's local name space provides a way to customize the user's view of the network. The services available in the network all export file hierarchies. Those important to the user are gathered together into a custom name space; those of no immediate interest are ignored. This is a different style of use from the idea of a 'uniform global name space'. In Plan 9, there are known names for services and uniform names for files exported by those services, but the view is entirely local. As an analogy, consider the difference between the phrase 'my house' and the precise address of the speaker's home. The latter may be used by anyone but the former is easier to say and makes sense when spoken. It also changes meaning depending on who says it, yet that does not cause confusion. Similarly, in Plan 9 the name /dev/cons always refers to the user's terminal and /bin/date the correct version of the date command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing date. Plan 9, then, has local name spaces that obey globally understood conventions; it is the conventions that guarantee sane behavior in the presence of local names.

The 9P protocol is structured as a set of transactions that send a request from a client to a (local or remote) server and return the result. 9P controls file systems, not just files: it includes procedures to resolve file names and traverse the name hierarchy of the file system provided by the server. On the other hand, the client's name space is held by the client system alone, not on or with the server, a distinction from systems such as Sprite [OCDNW88]. Also, file access is at the level of bytes, not blocks, which distinguishes 9P from protocols like NFS and RFS. A section by Welch compares Sprite, NFS, and Plan 9's network file system structures [Welc94].

This approach was designed with traditional files in mind, but can be extended to many other resources. Plan 9 services that export file hierarchies include I/O devices, backup services, the window system, network interfaces, and many others. One example is the process file system, /proc, which provides a clean way to examine and control running processes. Precursor systems had a similar idea [Kill84], but Plan 9 pushes the file metaphor much further [PPTTW93]. The file system model is well-understood, both by system builders and general users, so services that present file-like interfaces are easy to build, easy to understand, and easy to use. Files come with agreed-upon rules for protection, naming, and access both local and remote, so services built this way are ready-made for a distributed system. (This is a distinction from 'object-oriented' models, where these issues must be faced anew for every class of object.) Examples in the sections that follow illustrate these ideas in action.

The command set of Plan 9 is similar to that of UNIX. The commands fall into several broad classes. Some are new programs for old jobs: programs like Is, cat, and who have familiar names and functions but are new, simpler implementations. Who, for example, is a shell script, while ps is just 95 lines of C code. Some commands are essentially the same as their UNIX ancestors: awk, troff, and others have been converted to ANSI C and extended to handle Unicode, but are still the familiar tools. Some are entirely new programs for old niches: the shell rc, text editor sam, debugger acid, and others displace the better-known UNIX tools with similar jobs. Finally, about half the commands are new.

Plan 9 is characterized by a variety of servers that offer a file-like interface to unusual services. Many of these are implemented by user-level processes, although the distinction is unimportant to their clients; whether a service is provided by the kernel, a user process, or a remote server is irrelevant to the way it is used. There are dozens of such servers; in this section we present three representative ones.

Perhaps the most remarkable file server in Plan 9 is 8K2, the window system. It is discussed at length elsewhere [Pike91], but deserves a brief explanation here. 8V2 provides two interfaces: to the user seated at the terminal, it offers a traditional style of interaction with multiple windows, each running an application, all controlled by a mouse and keyboard. To the client programs, the view is also fairly traditional: programs running in a window see a set of files in /dev with names like mouse, screen, and cons. Programs that want to print text to their window write to /dev/cons; to read the mouse, they read /dev/mouse. In the Plan 9 style, bitmap graphics is implemented by providing a file /dev/bitblt on which clients write encoded messages to execute graphical operations such as bitblt (RasterOp). What is unusual is how this is done: 8V2 is a file server, serving the files in /dev to the clients running in each window. Although every window looks the same to its client, each window has a distinct set of files in /dev. 8V2 multiplexes its clients' access to the resources of the terminal by serving multiple sets of files. Each client is given a private name space with a different set of files that behave the same as in all other windows. There are many advantages to this structure. One is that 8V2 serves the same files it needs for its own implementation—it multiplexes its own interface—so it may be run, recursively, as a client of itself. Also, consider the implementation of /dev/tty in UNIX, which requires special code in the kernel to redirect open calls to the appropriate device. Instead, in 8K2 the equivalent service falls out automatically: 8V2 serves /dev/cons as its basic function; there is nothing extra to do. When a program wants to read from the keyboard, it opens /dev/cons, but it is a private file, not a shared one with special properties. Again, local name spaces make this possible; conventions about the consistency of the files within them make it natural.

Plan 9 runs on a variety of hardware without constraining how to configure an installation. In our laboratory, we chose to use central servers because they amortize costs and administration. A sign that this is a good decision is that our cheap terminals remain comfortable places to work for about five years, much longer than workstations that must provide the complete computing environment. We do, however, upgrade the central machines, so the computation available from even old Plan 9 terminals improves with time. The money saved by avoiding regular upgrades of terminals is instead spent on the newest, fastest multiprocessor servers. We estimate this costs about half the money of networked workstations yet provides general access to more powerful machines.

Plan 9 utilities are written in several languages. Some are scripts for the shell, rc [Duff90]; a handful are written in a new C-like concurrent language called Alef [Wint95], described below. The great majority, though, are written in a dialect of ANSI C [ANSIC]. Of these, most are entirely new programs, but some originate in pre-ANSI C code from our research UNIX system [UNIX85]. These have been updated to ANSI C and reworked for portability and cleanliness.

REFERENCES

[1] 16/32-bit lpc2000 family. http: //www.nxp.com /products /microcontrollers/32bit/index.html.

[2] ARM extended trace macrocell (etm) technical reference guide. http://www.arm.com/documentation/TraceDebug.

[3] ARM's coresight on-chip debug and trace technology. http://www.arm.com/products/solutions/CoreSight.html.

[4] Armulator, ARM. http://www.arm.com /support/ARMulator.html.

[5] Device file system guide. http://www.gentoo.org/doc/en/devfs guide.xml.

[6] exportfs, srvfs - network file server from plan9 man pages. http://plan9.belllabs.com/magic/man2html/4/exportfs.

[7] Features of the msp430 bootstrap loader (rev. d). http://focus.ti.com/lit/an/slaa089d/slaa089d.pdf.

[8] Freescale, MPC565 user's manual, 2002.

[9] Introduction to on-board programming with intel flash memory. http://www.intel.com/design/flcomp/applnots/29217902.pdf.

[10] Iso 13239 : High-level data link control protocol.

[11] Msp430 : Ultra low power mcu from texas intruments. http://www.ti.com/msp430.

[12] National ecological observatory network. http://www.neoninc.org.

[13] OCP-IP: Open Chip Protocol International Partnership. http://www.ocpip.org.

[14] Providing asynchronous file i/o for the plan 9 operating system. http://pdos.csail.mit.edu/papers/plan9:jmhickey-meng.pdf.

[15] Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap.

[16] Simulavr: an AVR simulator. http://savannah.nongnu.org .

[17] The two percent solution. http://www.embedded.com/story/OEG20021217S0039.

[18] What processor is in your product? http://www.embedded.com/columns/showArticle.jhtml?articleID=193101174.

[19] Emstar: A software environment for developing and deploying wireless sensor networks. In Proceedings of the USENIX 2004 Annual Technical Conference, 2004.

[20] The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, 2004. http://www.nexus5001.org.

[21] Guest editorial: Concurrent hardware and software design for multiprocessor SoC. Trans. On Embedded Computing Sys., 5(2):259–262, 2006.

[22] D. E. L. G. M. H. A. Cerpa, J. Elson and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In Proceedings of the 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean, April 2001, 2001.

[23] K. M.-M. A. Mayer, H. Siebert. Debug support, calibration and emulation for multiple processor and powertrain control socs. IEEE Trans. Comput., 55(2):174–184, 2006.

[24] C. G. A. S. Tanenbaum and B. Crispo. Taking sensor networks from the lab to the jungle. IEEE Computer Magazine, 39(8):98–100, 2006.

[25] D. F. Bacon. Realtime garbage collection. Queue, 5(1):40–49, 2007.

[26] T. W. Bart Vermeulen and S. Bakker. Ieee 1149.1-compliant access architecture for multiple core debug on digital system chips. In Proceedings of the International Test Conference, 2002.

[27] K. A. Bartlett, R. A. Scantlebury, and P. T.Wilkinson. A note on reliable full-duplex transmission over half-duplex links. Commun. ACM, 12(5):260–261, 1969.

[28] S. Bhattacharya, J. Darringer, D. Ostapko, and Y. Shin. A mask reuse methodology for reducing system-on-a-chip cost. In ISQED '05: Proceedings of the 6th International Symposium on Quality of Electronic Design, pages 482–487, Washington, DC, USA, 2005. IEEE Computer Society.

[29] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004, March 2004.

[30] Bluetooth.com : The official Bluetooth Technology Website. http://www.bluetooth.com/bluetooth/.