Protium, an Infrastructure for Partitioned Applications

Cliff Young, Lakshman Y.N., Tom Szymanski, John Reppy, David Presotto,
Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse
Bell Laboratories, Lucent Technologies
{cyoung,ynl,tgs,jhr,presotto,rob,girija,sape,ehg}@research.bell-labs.com

Abstract

Remote access feels different from local access. The major issues are consistency (machines vary in GUIs, applications, and devices) and responsiveness (the user must wait for network and server delays). Protium attacks these by partitioning programs into local viewers that connect to remote services using application-specific protocols. Partitioning allows viewers to be customized to adapt to local features and limitations. Services are responsible for maintaining long-term state. Viewers manage the user interface and use state to reduce communication between viewer and service, reducing latency whenever possible.

System infrastructure sits between the viewer and service, supporting replication, consistency, session management, and multiple simultaneous viewers. The prototype system includes an editor, a draw program, a PDF viewer, a map database, a music jukebox, and windowing system support. It runs on servers, workstations, PCs, and PDAs under Plan 9, Linux, and Windows; services and viewers have been written in C, Java, and Concurrent ML.

1 Introduction

In the 1970's, one could walk up to any telephone in the world and use it as easily as one's home telephone. The computer revolution might have followed suit, but the opposite holds. It is nearly impossible to use the neighbor's computer: data files are unavailable or not available in a consistent place, the wrong applications are installed, and the preferences for those applications are personalized.

Why can't one use any computer on the planet? The reasons are historical and economic. First, the last twenty years of the computing industry have been about *personal* computing. Mainframe and minicomputer users shared a consistent (if sometimes unresponsive) environment in their single, shared system. In contrast, every PC is customized, binding large amounts of state to each PC. The move to PCs gave users responsiveness at the cost of consistency. Secondly, networked computing environments work "well enough" within a single security domain such as a corporate intranet or university-wide network. Such environments do allow users to log into multiple terminals, and this covers a large percentage of shared-use cases. Third, remote access tools are "good enough": users are willing to dial into or tunnel to their corporate intranet to get access, despite added latency or inconsistency. Lastly, the Internet grew up only in the last decade; before that one couldn't think of being connected to every computer in the world. A new generation of computing devices is upon us; each person will have many devices and each person will expect the multiple and remote devices to work consistently. We will have to rewrite all of our applications for the new devices anyway; why not rewrite them so they work better?

Our goal is to be able to use any Internet-connected device as if it were the machine in our office. Further, we want this access consistently and responsively. Consistency is similarity of experience across devices. The user's session must migrate from device to device. Each application will adapt to each end device so that it exploits the device's unique capabilities and works around the device's limitations. Responsiveness implies that remote access is as comfortable as a local application. Many remote access systems have addressed one or the other of these two goals; few address both.

Protium splits applications into two pieces. One runs near the user; the other runs in a service provider that is highly available, has persistent storage, and has abundant computation cycles. We call these pieces *viewers* and *services*, respectively, to emphasize that state is maintained by the service. Viewers and services communicate via an *application-specific* protocol; the application designer must partition the application to maximize consistency and responsiveness. Applications are built as if only a single viewer-service pair existed, with certain additional constraints. These constraints allow the Protium infrastructure to support connection, reconnection, multiple simultaneous viewers, state consistency and replication, and session management.

The Protium prototype includes a text editor, a MacDraw-style drawing program, a PDF viewer, a map viewer with map database, a digital music jukebox, and windowing system support. Viewers and services have been written in C, Java, and Concurrent ML and run under the Plan 9 Operating System, inside Java applets, and under Linux. All of these viewers and services interoperate.

To appear in the Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII). Copyright © IEEE 2001.

2 A Better World

We'd like to be able to work all day at the office, using a research operating system such as Plan 9 or Linux. At the end of the day we'd like to walk away from the office computer, possibly with state uncommitted to stable storage. On the train home, we'd like to be able to pull out a wireless PDA or cellular phone, and have the portable device replicate the office session. PDA-specific viewers for each of our applications will be launched that show the exact same state, including uncommitted changes, as the work session back in the office. The PDA is limited, but one could imagine reading drafts of a document or fixing typos even on its small screen and using its limited input capabilities. When we get home, the home computer runs only Windows. But using a web browser and Java applets we again replicate the session, getting Java versions of each of our applications with the same session state as we had at the end of our train ride. In each remote case (PDA and Java), the applications respond immediately to user input; updates spool back to the office server.

Our two consistency-related goals are *session mobility* and *platform independence*. The example shows session mobility where the user accessed the same state of applications using three different systems. We will not supply a precise definition of "session" in this paper; however, an intuitive definition would be the state of all applications currently open on one's workstation screen. Platform independence involves accessing the same session on a variety of devices and operating systems: a workstation running a workstation OS, a PDA with its proprietary OS, and a Windows home PC with a standard browser. And to make things difficult, we will not sacrifice responsiveness for these consistency goals.

This example sounds like typical ubiquitous computing propaganda, but our system concretely provides them: we have a prototype system running. We next describe the assumptions about future technologies that underlie our engineering choices, then go on to describe our approach and prototype system. Before concluding, we explain why prior approaches fail some of our requirements.

3 Assumptions

We make some assumptions about the future. On the technical front, Moore's law continues, exponentially improving processing power, memory sizes, device sizes, heat dissipation, and cost. As a corollary, the world will move to multiple devices per person. Bandwidth will increase in the backbone network and will increase (albeit less quickly) to portable and home devices. Wired or wireless remote coverage will improve over the next decade; we thus choose not to focus on disconnected operation. However, we also assume that communications latency will *not* improve much in the next decade.

While the speed of light places a fundamental lower bound on communications, it takes light only about 130 milliseconds to go around the world. Our latency assumption instead rests on the current realities in data networking, where differentiated services have not yet been deployed and switching delays are significant. Even with a speedy core Internet, however, it seems believable that last hop communications services would still experience notable delays for the next decade (today, we regularly experience *10 second* round-trip times on CDPD modems and WAP cellular phones; systems must respond within 100 milliseconds to feel instantaneous and within 1.0 second not to disrupt the user's flow of thought [3]). Furthermore, 130 milliseconds is very long for a computer, so services that rely on other services still face latency issues.

On the social front, we have two assumptions. First, we believe that there will be a new round of operating system wars for the PDA/cellphone market. The market will determine which (if any) of the current contenders (PalmOS, Windows CE, Psion, to name a few) will win. In the meantime, we should deploy systems that work well regardless of programming language or operating system.

Our second social assumption is that distributed programming is hard. If we can find ways to sweep many of the traditional distributed programming problems under the rug of our infrastructure, the average programmer might be able to write a robust distributed application. A secondary goal is that writing an application in our system will not be much harder than writing a standalone GUI-based application is today.

4 Partitioned Applications

Our approach draws its inspiration from two applications that work when a low-performance channel connects the user and his data: the Sam text editor [4] and the IMAP mail protocol [1]. Sam comes in two pieces. Sam's service runs near the file system where editing takes place; Sam's viewer runs on whatever device the user has at hand. Viewer and service communicate using a protocol that keeps track of the state of both halves. IMAP works similarly but for mail instead of editing. Both applications divide the task into two parts: a service that is highly available and has large compute and storage resources, and a viewer that needs a connection to the service and some kind of user interface but need not download the entire program state. Perhaps the central question of our project is: can we generalize from Sam and IMAP to all applications? And can we build infrastructure that makes this easy? We call this approach, "partitioned applications," because the network breaks the application into parts.

Another way of looking at Protium is that we are "putting the network into the application." Most previous approaches divide remote from local at an existing abstraction layer, for example the file system, the GUI API (The X Window System [6]), or the frame buffer. Partitioning incorporates these prior approaches; it just adds a new dimension of flexibility.

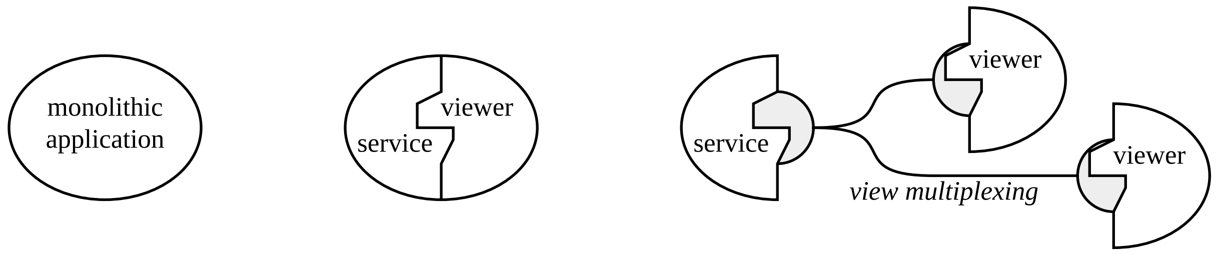


Figure 1: A monolithic application, the same application after partitioning, and multiple viewers connected to a single service through a view multiplexer.

Partitioning induces rich systems issues. For example, what manages a session? How does one connect or reconnect to a service? What maintains consistency, replicates state, or provides multicasting across simultaneously active viewers? What happens when a viewer crashes or the viewer device is lost? Our prototype system suggests preliminary answers to each of these questions.

5 Prototype System

Our prototype system currently supports five applications in addition to the session service/view manager. These are a simple text editor, a MacDraw-style drawing program, a PDF image viewer, a map program (with both photographic images and polygonal graphics), and a music jukebox. These represent a variety of interesting desktop applications, so we are encouraged that we have been able to build them with our current infrastructure. However, for us to really claim that we are building a general system, we need to build more applications. We are investigating video and hope to add PDA applications (calendar, email, address books) to our suite.

One of the most interesting research issues involves adapting viewers to the platform on which they run. There are at least four different kinds of platforms: big bitmaps (desktops and laptops), small bitmaps (PDAs and cellphones), text, and voice. We present some preliminary results about device-specific adaptation in the section on session management, but this topic remains largely untouched.

Building applications around a protocol gives us a high degree of language and operating system independence. Our prototype applications run under Plan 9 (all viewers and the edit and draw services), Java (all but the PDF viewer; map and juke services), and Linux (draw and PDF services). The Plan 9 programs were written in C; the Linux programs were written in Concurrent ML. Porting remains a significant task, but this wide variety of languages and systems supports a claim to language and OS independence.

Just rewriting applications into two pieces doesn't make a systems project. For Protium, the interesting issues are in the infrastructure, and we describe two pieces of the infrastructure here, followed by an application example. The first piece of infrastructure, the view multiplexer, supports multiple viewers on a single service, while simulating a connection to a single counterpart to each viewer or service. The second piece of infrastructure, the session service, bundles together multiple services into a session; it has a corresponding piece, the view manager, which runs on the viewing device. After describing the pieces of infrastructure, we will go on to an application example, our map program.

5.1 Multiplexed Viewers

Each viewer or service is designed as if it spoke to a single counterpart service or viewer, respectively. But we want to be able to support multiple viewers simultaneously connected to a single service. The view multiplexer simulates a single viewer to a service. New viewers that wish to connect to a running service do so through the view multiplexer, so the service need not be aware that a new viewer has connected. Figure 1 shows a view multiplexer interposed between a service and multiple viewers.

To do its job, the view multiplexer snoops the messages between service and viewer. Each message in the system has a tag to help the multiplexer. Most communication is synchronous from viewer to service, in viewer-initiated request-response pairs. Viewers can generate read, lightweight write, and heavyweight write messages. Services respond with either acknowledgements (ACKs) or negative acknowledgements (NACKs); the infrastructure is allowed to NACK a message without allowing the message to reach the service. Viewers must also be able to handle asynchronous update messages, which are generated by the infrastructure when one viewer receives an ACK; an update tells a viewer that some other viewer succeeded in updating the state. Since all viewers see all ACKs, they can keep their views of the state up-to-date. Lastly, services can asynchronously broadcast to all viewers; broadcast messages support streaming media. This consistency model is similar to publisher-subscriber consistency models.

In addition to multicasting ACKs (as updates) to all viewers, the view multiplexer helps build responsive viewers. Using a simple token-passing scheme, the view multiplexer allows one viewer to become privileged. Lightweight writes from the privileged viewer are immediately ACKed; this allows the privileged viewer to deliver local response time. These writes must then be propagated to the service and the other viewers; formal

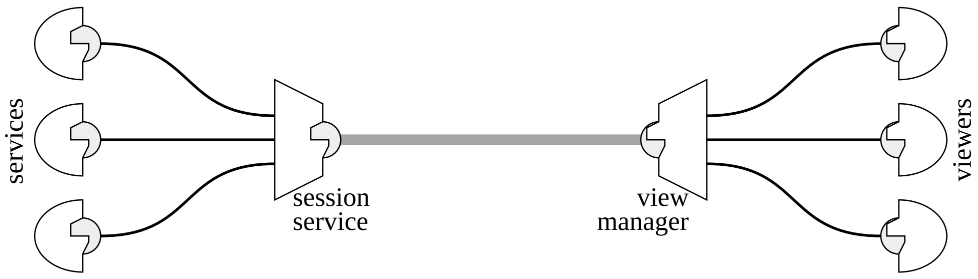


Figure 2: Three applications (with service and viewer pieces) connected through the session service/view manager multiplexor/demultiplexor pair.

requirements of the protocol and the service implementation guarantee that the service will acknowledge the write. Lightweight writes should be common actions that do not require support from the service, e.g., responding to keystrokes or mouse events. Global search-and-replace or commit to stable storage should be heavyweight writes. Token management matches our intended uses, where a single user expects immediate response from the device he uses but can tolerate delayed updates in other devices.

5.2 Session Management

Intuitively, a session is the state of one's desktop. The Protium *session service* runs on the service side and bundles together multiple services into a single session. A device-specific *view manager* connects to the session service and runs the viewers that correspond to the services in the session. The session service and view manager form a multiplexer/demultiplexer pair, linking multiple viewers to multiple corresponding services. In our introductory example, the view manager launched the viewers on the PDA and under the Java-enabled browser.

The session service and view manager behave as another application pair, so the pipe between them can be managed by the view multiplexer just like for any other application. The session and view managers can also hierarchically encapsulate and route messages for the underlying service/viewer pairs; however, services and viewers are free to communicate out-of-band if the designer so chooses. Figure 2 depicts a set of applications managed by a session service and view manager. We can compose view multiplexers and session/view manager pairs in arbitrary nesting; the two sets of multiplexers recurse.

In addition to managing the set of applications, the view managers adapt window system events to their devices. Moving or resizing a window on a big screen causes a corresponding move or resize on another big screen. Moves and resizes have no small screen analog (because applications typically use the whole screen); however, focus changes and iconification work similarly on big and small screens. We have not yet implemented the text-only or voice-based viewers, so we have no experience in this area; a text-only view manager should work like a shell. Another open problem is adapting to differently sized big screens.

5.3 A Protium Application: Map

Some of us (Szymanski and Lakshman) are interested in geographic data such as terrestrial maps, aerial images, elevation data, weather information, aviation maps, and gazetteer information; one goal is synthesizing these views coordinated by positional information. Gigabytes of data come from scattered sources and multiple servers. We had built a Java geodata viewer that could navigate and edit the data. Different types of geodata show as selectable display layers. Different layers or different parts of the same layer may be served by different machines. However, all servers and the viewer have the same abstract view of the data and hold some piece of the data locally. This uniform view results in a simple protocol between the viewer and the servers.

We used the Protium infrastructure to share the geodata viewer, the basic idea being that multiple viewers can share a session, moderated by a session server, that tracks what is being viewed and tells viewers what data to get and from where. Actual data travels out-of-band; only control messages route through the Protium infrastructure. The session server also supports textual messaging so that a shared viewer can be used to give driving directions to someone (with appropriate maps and messaged instructions) at a remote location.

The map application is designed to hide latency from the user. For example, the viewer displays street names and addresses in a tool tip; a remote query would make this feature too slow and too variable in latency. All geographical data is kept in a tiled format, compressed using a method appropriate to its type, and transmitted to the viewer upon demand or (sometimes) before. The map viewer stores this data in a two-level cache in which the lower level (which counteracts network latency) contains compressed data, and the upper level (which counteracts decompression latency) contains fully expanded data structures needed to support user interactions. Requests to map services are executed in batches that are satisfied in an out-of-order fashion. This overlaps server processing with both network transmission time and client decompression time. This architecture provides a degree of responsiveness that could not be approached with a conventional browser/server structure.

As part of this exercise, we implemented a new viewer in C under Plan 9. The Java and Plan 9 viewers differ greatly: the Plan 9 viewer targets the small but colorful iPAQ display and uses pen input; the Java viewer runs on big screens and uses the real estate for a complex GUI.

The session server (excluding marshalling code) is about 330 lines of Java. An additional 286 lines allow the existing viewer to interoperate with the session server. The Plan 9 viewer required 3527 lines of C of which about 300 lines deal with communication and the rest deal with graphics and event handling. Thus, with a small amount of effort, we were able to convert an existing single-user application (which was already split into service and viewer parts) into a shared application.

6 Remote Access

The body of related work is far too vast to survey in a position paper; remote access systems span many disciplines including operating systems, networking, distributed systems, and databases. This section instead highlights major approaches to remote access and explains why they do not meet our goals.

Most remote access systems fail one or both of our consistency and responsiveness requirements. Rlogin and its more secure modern descendant, ssh [7], are platform independent but do not provide session mobility. Distributed file systems (examples abound; AFS, Coda, and Locus/Ficus to name a few) allow one to access stable storage wherever the network reaches but say nothing about how to provide applications. Remote Procedural Call packages similarly do not show how to provide applications. Distributed object frameworks such as CORBA and DCOM address the same problem as we do, but suffer performance problems because their abstraction of remote and local objects hides latency from designers [8]. Recent work in thin-client computing and its predecessor, client-server computing, give some forms of consistency but force clients to wait during both network and server latencies.

A number of systems apply the brute-force approach of sending screen differences and raw user input information across the network. Examples include Virtual Network Computing (VNC) from AT&T Research [5], the SunRay product from Sun Microsystems, Citrix System's Windows-based product, and Microsoft's NetMeeting. All of these systems provide bit-for-bit consistency but suffer when network latency increases. They also do not adapt to the constraints of local devices: viewing large virtual screens on small physical devices is difficult, and the system architecture prevents further device-specific adaptation.

Philosophically, the Berkeley Ninja project is closest to our approach [2]. We follow Ninja's approach of keeping stable storage in a service provider (Ninja calls this a "base") and allowing "soft" state in the viewers to be lost. Ninja focuses on scalable services; Protium focuses on the applications we use daily on the desktop. Protium's infrastructure works primarily between service and viewer.

7 Experiences

To partition an application, one must focus on the application-specific protocol. We would like to present a how-to guide on partitioning, but the lessons so far sound like platitudes, including "match messages to user input", "separate control and data", and "beware round trips." In an early version of the draw protocol, each object deleted required a separate message. If the user selected a number of objects and issued a delete command, some of the deletes might fail, leaving the application in a confusing state. The juke, map, and PDF applications have large data streams (music, graphics/image, and image); waiting for a large object to be transmitted can keep a small control message from taking effect. Designing protocols without a delay simulator is dangerous: what works acceptably on a LAN may be unusable with a 1-second round-trip time. We hope to be able to summarize and illustrate more such principles in the future.

Protocol designers must decide which application state is viewer-specific and which is service-mediated. For example, the edit application keeps scroll bar position and text selection local to the viewer. The draw application tries to do the same, but some operations (grouping, ungrouping, and deleting objects) reset the selection to be consistent across all viewers. More ambitiously, it might be useful to be able to preview the next PDF page on one's remote control device while continuing to show the current page on the video projector device, but this is not yet supported by the PDF protocol. Some applications have added state expressly for collaborative or remote-control purposes: the map and PDF programs both support telestrator-style overlays, and the map program also includes a chat room.

Writing viewers is harder than we would like. Viewers include all of the state and complexity of a traditional stand-alone application, augmented by the complexity of managing a single outstanding request while being able to accept asynchronous updates and broadcasts. Backing out attempted changes when a NACK arrives further complicates design. All of our viewer programs are multi-threaded; this seems a higher standard than we would care to impose on the average programmer. We are exploring programming idioms, APIs, and library support that might simplify viewer development.

8 Discussion and Conclusion

This system is not about the next killer application. If anything, we are rebuilding all of our old applications to live in a new world. This follows our biases as system builders: we know how to build infrastructure. If this project or one like it succeeds, we will have universal data service, like universal telephone service. The new devices require rewriting all of our old applications anyway. We might as well get some benefit out of it.

The Protium approach makes additional demands on application programmers. The initial designer of an application creates an application-specific protocol, while designers of new viewers or services must adhere to that protocol (if our project succeeds, then perhaps standards for application protocols will emerge). Porting an application to a new platform involves at least porting the viewer. Building a viewer combines both traditional GUI issues and communicating back to the service. Services may also need to be ported.

What is the best way for Protium to support existing applications? It depends on the application. Programs with clean separation between display and state integrate easily with Protium; most programs, however, are large, complex, and have tangled state- and display-management code. We observe that the move to new devices such as PDAs and phones will force such programs to be rewritten anyway; integrating the program with Protium as part of the rewrite will be a modest extra requirement and will benefit the application by making it use the network more effectively.

Considering applications in a partitioned context provides new opportunities to use old tricks. Persistence is a service-only problem; the service need not worry about geographic distribution, so known persistence techniques apply. Viewers that are lost or lose state are easily replaced or restored because the service is the repository of record. The connection between service and viewer can be a network socket; known techniques for authentication and encryption therefore apply. Security, logging, caching, and prefetching seem like obvious features to add. This paper concentrates on the single user; Protium also gives limited support for collaboration and remote control. We think of these as bonuses rather than our primary research goal; it seems a high enough goal to be able to use any computer in the world.

Protium is the most common isotope of hydrogen, the most common element in the universe. A protium atom has two pieces that are closely coupled and essential to the nature of hydrogen, but the two pieces are different from each other. And while the two pieces are themselves basic, the exploration of their interaction has occupied scientists for more than a century.

9 References

[1] M. Crispin. Internet Message Access Protocol – Version 4rev1. RFC2060 (December 1996).

[2] S. D. Gribble, et al. The Ninja Architecture for Internet-Scale Systems and Services. To appear in a *Special Issue of Computer Networks on Pervasive Computing*.

[3] R. B. Miller. Response time in man-computer conversational transactions. *Proc. AFIPS Fall Joint Computer Conference*, 33:267–277, 1968.

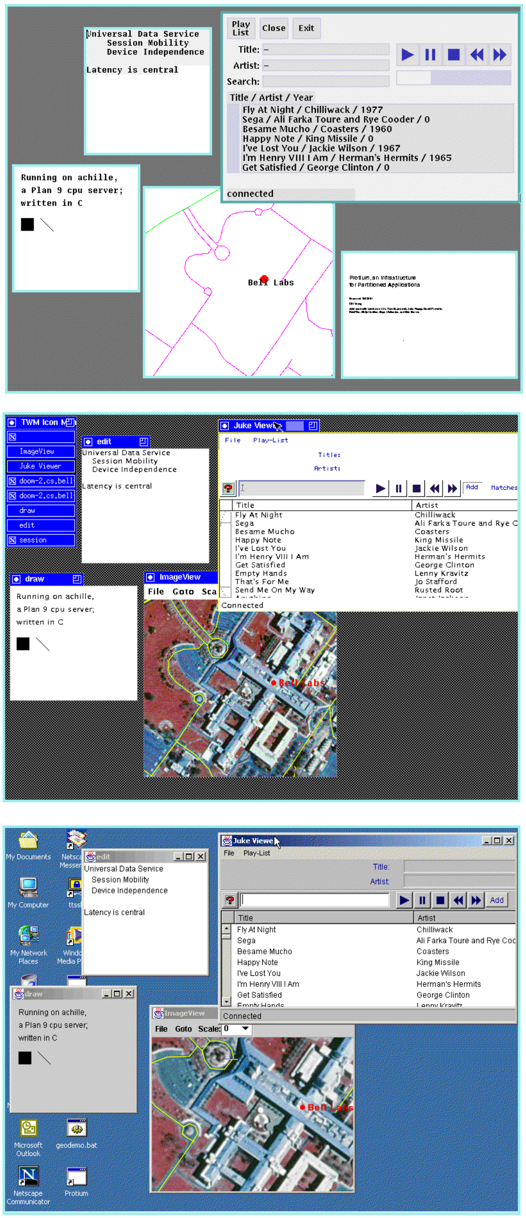
[4] R. Pike. The text editor Sam. *Software Practice and Experience*, 17(11):813–845, 1987.

[5] T. Richardson, et al. Virtual Network Computing. *IEEE Internet Computing*, 2(1): 33-38, Jan/Feb 1998.

[6] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–106, Apr. 1986.

[7] T. Ylonen. The SSH (Secure Shell) Remote Login Protocol. In *Internet Drafts* (November 1995).

[8] J. Waldo, et al. A Note on Distributed Computing. In *Lecture Notes in Comp.Sci. 1222*, Springer, 1997.



```
Universal Data Service
    Session Mobility
    Device Independence

Latency is central
```

```
Running on achille,
a Plan 9 cpu server;
written in C
```

Protium, an Infrastructure for Partitioned Applications

```
Title / Artist / Year
Fly At Night / Chilliwack / 1977
Sega / Ali Farka Toure and Rye Cooder / 0
Besame Mucho / Coasters / 1960
Happy Note / King Missile / 0
I've Lost You / Jackie Wilson / 1967
I'm Henry VIII I Am / Herman's Hermits / 1965
Get Satisfied / George Clinton / 0

connected
```

Title	Artist
Fly At Night	Chilliwack
Sega	Ali Farka Toure and Rye C
Besame Mucho	Coasters
Happy Note	King Missile
I've Lost You	Jackie Wilson
I'm Henry VIII I Am	Herman's Hermits
Get Satisfied	George Clinton
Empty Hands	Lenny Kravitz
That's For Me	Jo Stafford
Send Me On My Way	Rusted Root

Connected

Figure 3: Screen shots of Protium applications under Plan 9 (top), Linux/Java (middle), and Windows/Java (bottom). Within each screen, the applications are edit (top left), draw (left), juke (top right), map (bottom), and PDF (bottom right, Plan 9 only).