Distributing Processing without DPEs: Design Considerations for Public Computing Platforms

Timothy Roscoe and Bryan Lyles
Sprint Advanced Technology Labs, 1 Adrian Court, Burlingame, CA 94010, USA
{troscoe,lyles}@sprintlabs.com

1 PUBLIC COMPUTING PLATFORMS: INTRODUCTION AND MOTIVATION

What is a Public Computing Platform? A Public Computing Platform is a system of hardware and software which provides a computing environment for a number of paying customers, some of whom may be competing with each other for business. Typically these customers will be providers of Internet services, who are themselves generating revenue from their service (for example, by charging end users or providing advertising) and paying the provider of the Public Computing Platform to host their service.

The work described in this position paper rests on a number of assumptions. The first is that Public Computing Platforms will become a reality. The second, more interesting, assumption is that there will be a compelling business case for such platforms to host many more services than there are physical computers making up the platform. Systems that host services with at least one processing node per application are relatively simple to build and are in widespread deployment today.

Designing, building, and operating Public Computing Platforms clearly poses a number of engineering challenges. For one thing, the platform must be highly available and reliable: businesses will depend on it for their revenue. It must also provide excellent scalability: the market for services is expected to grow, and like any other computer applications we can expect the resource demands of individual Internet services to grow also. The platform must therefore scale in performance in two dimensions, both to handle a potentially large number of services at once, and also scale over time as the resource demands of a given set of services increase. This introduces a new requirement of incremental deployment: it must be possible to add capacity to the platform without interrupting service.

Two further requirements set Public Computing Platforms apart from more traditional systems: security, and resource isolation. The platform must support many services, not all of which can be equally trusted, and some of which may even be competing with each other. This implies that the state of one service must be secured from unauthorised access or modification by another service. Additionally, a service must not be permitted to use resources allocated to, required by, and paid for by, another service.

A Public Computing Platform can be contrasted with what is sometimes called an *Enterprise Computing Platform*. An Enterprise Computing Platform (of which there are many in deployment) is designed to run a single application (or a very small number) on behalf of a single enterprise with high capacity (throughput, etc.) and availability. Examples of such platforms are web portals, search engines, transaction processing systems, database servers, mail servers, media servers, etc. We can identify two different kinds of players involved with Enterprise platforms:

1. The Enterprise itself. There is one enterprise per platform. The enterprise owns the platform and the application. Typically the platform is entirely dedicated to running the enterprise application. This is the case even when the hosting function is outsourced, and the hardware platform is owned and operated by a hosting entity.

2. The application users. For some transaction processing applications (for example), the users will belong to the enterprise itself. For others (WWW-based businesses), the users are outside the enterprise.

In contrast, a Public Computing Platform can be expected to be running a large number of services simultaneously. Consequently, the players involved are different, and we can identify three different kinds:

1. The Application Providers. Unlike the enterprise case, there are many application providers utilizing the same platform; some may even be in competition with each other. Application providers pay the platform provider for computational and network resources they use in providing their service.

2. The Platform Provider. The platform provider owns the platform on which the services run. Unlike the current case of a simple hosting service, platform providers must manage many mutually untrusting applications per

machine, provide resource guarantees to the application providers, bill them accordingly, and rapidly change application resource allocations if the need arises.

3. The Application Users. These end-users are the same as the enterprise case, except that now they are collectively using a large number of different services rather than the one.

The technological problem for the Platform Provider is complicated by the need for Application Providers to replicate services across the platform as a whole, for reasons of availability and fault tolerance.

A number of groups are working on platforms for Internet services (for example, [7, 2, 9]). All share the characteristic of running services over a distributed processing environment or middleware platform, which provides a uniform API and communication subsystem.

We believe such an approach is fundamentally *wrong*.

2 A POLEMIC AGAINST MIDDLEWARE

There are many distributed processing environments, sometimes called distributed computing environments or middleware platforms, in use today. The line between DPEs and platforms for distributed services is also sometimes blurred.

However, most Internet services (Web, mail, directory and file servers, for example) are written to run over an OS (POSIX, Windows NT, etc), rather than over a distributed processing environment. Modulo the usual gripes, they seem to work fine. More importantly, they do not appear to suffer from the absence of a middleware platform or distributed processing environment.

Most of these services are much younger than the basic design principles behind distributed processing environments (which have been around for at least 15 years). Why, then, do new services not use middleware?

The negative consequences of distributed middleware

Pedlars of Internet middleware platforms propose that such services should in fact be reimplemented not over the OS but over a middleware layer which sits between the application and the OS. Such an approach has a number of serious disadvantages, none of which are in much dispute:

First, developers of new applications must write to the new programming interface, and use the new conceptual objects that the platform imposes. This may be a significant learning curve. Almost all distributed programming APIs look superficially the same, but the devil is in the details.

Second, programmers have to contend with a runtime bottleneck. A large body of software interposing itself between OS and application imposes a runtime overhead. It is unlikely to improve performance except by caching effects, which are generally better implemented in a more intelligent application-specific way (i.e. in the application), or in the OS.

Third, programmers also now have to contend with what might be termed a "semantic bottleneck" between the application and the OS. As a consequence of the DPE or middleware interposing itself between the two, some OS and hardware functionality may be hidden. Similarly, application design and functionality is constrained *a priori* by the design whims of the middleware producer. Aside from limiting functionality, the additional performance constraints this places on service implementors probably dominates the runtime overhead.

Fourth, the semantic bottleneck also constrains the evolution of the system over time, since the OS feature set exposed is frozen. The system can't evolve. It can't take advantage of either new OS features or new ways of conceptualizing applications.

Fifth, the middleware platform does nothing for existing applications, which are not written to use it. Middleware vendors have consistently failed to persuade developers of widely deployed applications to use their kits, instead deployment of middleware is generally restricted to large, one-off intra-enterprise applications. This point alone is a powerful argument against basing a Public Computing Platform on middleware.

These are powerful arguments against the use of middleware to develop a wide market in publicly accessible services, but so far apply to communication between the application and OS. When the middleware platform becomes a distributed processing *environment*, interposing itself between distributed components of applications, things get a lot worse. Two more factors appear:

First, by trying to standardise communication between applications and between distributed components of single applications, the middleware creates an additional semantic bottleneck of communication: invariably the type system is too restrictive, or RPC semantics inadequate. Many middleware systems (Ninja for example) have recognised this and sought to correct it by relaxing the invocation semantics and reducing the expressivity of the type system, without realising that the problem stems from believing communication should be standardised at a higher level than the OS in the first place.

Second, by trying to standardise interfaces and semantics over heterogeneous platforms in the name of location transparency, the middleware erases many of the positive *benefits* of heterogeneity: different applications with different requirements should be able to make use of different functionality.

Arguments in favour of middleware

Arguments are, of course, made in favour of using middleware systems. The benefits of middleware and distributed computing platforms are generally claimed in two areas: in-

teroperability, and abstraction.

Interoperability between applications has its uses, but it must be pointed out that middleware does not provide interoperability other than at a level of wire syntax and (at best) naming. Much application logic is required before interoperability between applications can actually achieve useful goals. Either way, there is a world of difference between tools or components that applications might employ in order to communicate, and an entire distributed processing environment inside which applications must be embedded in order to communicate. Note that today's Internet applications which are not built over DPEs achieve interoperability through standards built by consensus following an understanding of particular application spaces, rather than mandated in advance.

The provision of abstractions suffers from the problems discussed above. Which abstractions to provide? Why? How can one be so sure one has the right set? Our view is very much in line with that of Dawson Engler and Frans Kaashoek with regard to operating systems abstractions[4]: system-enforced abstractions prevent application programmers getting a useful picture of the runtime environment. Developers should be free to choose the abstractions which best suit their application.

We would add two refinements to their argument:

Firstly, any system will inevitably provide some abstractions. An abstraction is a choice as to what elements are deemed important and are to be made explicit, and which are deemed unimportant and can be safely ignored. Abstraction is inherent in engineering tradeoffs, and in any mechanism for sharing resources.

Secondly, abstractions are an important tool for understanding aspects of complex systems. System design would be impossible without them. This does not mean, however, that design abstractions must then be uniquely reified in code.

This last process appears to be what happens in middleware: there is a sinister, creeping tendency for the design of the middleware platform, which after all should simply be an aid to writing better distributed systems, to expand to the point where applications are almost subordinate to it, their form being largely determined by the particular distributed processing environment in force.

The difference between abstractions and implementations

Perhaps uniquely among distributed processing environments, the ANSA project [19] first defined a sophisticated conceptual model of distributed computation (which later fed into the ISO ODP reference model). The software realisation, ANSAware, was carefully described as one of many possible mappings of ANSA concepts into engineering artifacts [11][1].

[1]Having said that, ANSAware as an implementation suffered from the same tendencies we mention here

There is a *mapping* between abstract models used in thinking about a distributed system, and the implementation artifacts (software and hardware) that are built to realise the system. It is the mapping that is important when building the system, and this mapping may largely cover existing functionality of the OS, network and hardware, without the need for much additional implementation.

However, for some reason, traditional distributed systems, including cluster-based services, move from design abstractions to implementation by building completely new implementation objects which correspond exactly to the design abstractions. This approach comes to view the DPE as the only ground on which to build applications. The DPE becomes more important than the applications. The results are "take-over-the-world" schemes, with all their impracticality.

In summary

This problem with the reification of abstractions applies to *all* existing middleware platforms. It is commercially and politically unreasonable, as well as technologically bogus, to require that services running on a Public Computing Platform, written and operated by many third-party service providers, access the hardware through some new programming interface (or, indeed, through any *one* programming interface). It is further hubris to claim that this interface meets the needs of applications, past, present and future.

Distributed middleware is useful, but it is never universal. A general-purpose platform should never be based entirely on it.

3 A REALISTIC APPROACH

We are building a prototype Public Computing Platform at the Sprint Advanced Technology Lab. Like most such systems, our platform uses a cluster of computers in a controlled physical environment, for reasons of scalability and reliability.

However, our Public Computing Platform is not a middleware platform. If a particular Service Provider wishes to use a piece of middleware (for some reason) to implement their service this should be allowed, but the environment the platform provides to Service Providers is, simply, an operating system. Our system therefore differs from others in this space at a deep philosophical level: our API for each processing node is chosen to be that of the system software running on that machine. That system software, in turn, is a commodity OS (or something very like it). The contrast is illustrated in figure 1.

Philosophically, our platform is more closely related to recent projects in network and O/S resource control than to most other service platforms. Operating systems like Nemesis[12, 14], Scout[13] and Exokernel[5] provide a very low-level and explicit view of physical resources to the application, which then uses libraries to implement its own ab-

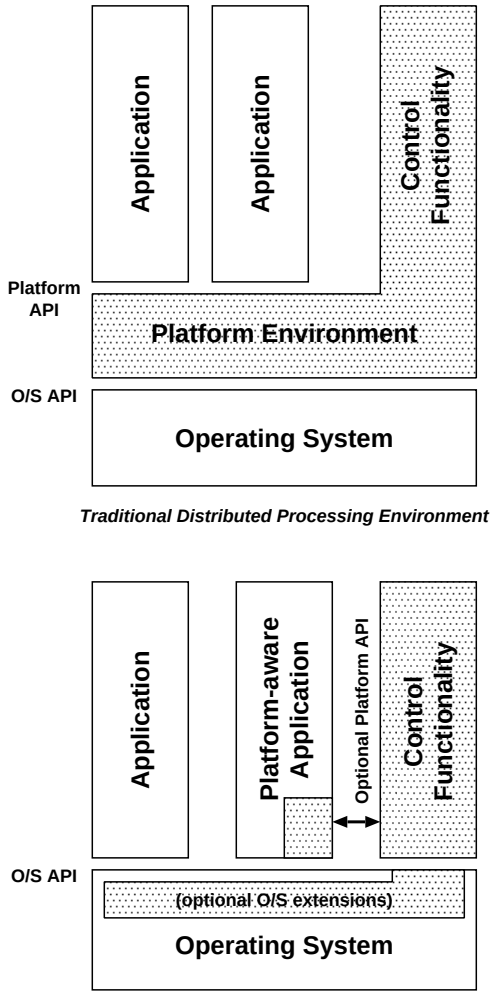

Figure 1: The contrast between a Distributed Processing Environment (DPE) approach to Public Computing Platforms (top), and our approach (bottom). The latter avoids imposing an additional set of abstractions between applications and the OS (thereby hiding information about the system), while still providing access to platform functionality for platform-aware applications.

stractions. The Xenoservers[15] project is one of several that seeks to apply some of these ideas to Internet services. An earlier, important OS development was the KeyKOS system[8], which had as an explicit aim support for mutually antagonistic applications and used a virtual machine monitor approach. Other, more restricted approaches exist (for example, resource containers [3] and QLinux [18]) which instead have the benefit of working with existing mainstream OS structures. Similarly, open signalling approaches in network such as Switchlets [16] and xbind [10] leave much signalling policy to end users, restricting their own abstractions to those for partitioning switch and network resources.

The philosophy behind these systems might be viewed as follows. We start with a set of agreed-upon givens. In the case of operating systems, these are the physical resources plus hardware mechanisms which can be used to multiplex them. In networks, these givens are switches and switch dividers. Then, we present to users a view which is as close to these givens as possible, subject to the requirements of sharing and security. Finally, we provide *non-mandatory* mechanisms and abstractions in the form of libraries to aid in the implementation of application-specific functionality.

Our givens for a public computing platform are the network, hardware, and OS software (and APIs) available. This is therefore what we present to users, subject to the security requirements of the platform and specific SLAs. Above this, users can run their own DPEs, libraries, etc. and in addition take advantage of the fault monitoring and availability mechanisms our infrastructure can provide, *without any of it getting in the way*.

System abstractions and implementation

We briefly describe the abstractions in our platform. We must emphasize that many of these abstractions are solely tools for thinking about the system, and in no way imply that we are building software objects corresponding to them. As far as possible, we map our abstractions onto *existing* system facilities. This has the important consequence that existing applications (Apache, for instance, or a QuakeIII server) fit into our architecture without recompilation. Our terminology is borrowed from ODP[1].

We call processing machines in our platform *nodes*. A node runs some operating system. Applications, owned by third-party clients of the platform run on one or more nodes, which they may share with other applications. We use the term *capsule* to refer to that portion of an application running on a particular node. A capsule will typically map to a process or process group, together with a set of resource containers or quotas, the exact nature of which will depend on the capabilities of the operating system.

As well as running capsules, each node also runs a *nucleus*, which is responsible for management of the individual node.

A nucleus can be viewed as an extension of the operating system functionality on the node to provide remote control of the node. The nucleus is responsible for starting, stopping, and monitoring capsules running on the node, and also for altering the allocation of node resources to capsules, using whatever facilities the node OS provides in this regard.

Since the programs executing within capsules are simply applications written for the OS in question with potentially no knowledge of the platform, we cannot mandate a standard, well-defined control interface between the nucleus and each capsule. Instead, we implement a capsule-specific control

mechanism within the nucleus by means of a domain-specific scripting language. The scripts execute in the context of the nucleus and in general emulate the actions that a human operator would perform to install, monitor, start and stop the service. This allows us great flexibility and evolvability in how we interface the control operations of the nucleus to the host OS and each class of application.

The nuclei on all the nodes on the platform are coordinated by a logically centralised management function called the *control plane*. The control plane handles platform-wide issues such as node failover, resource allocation, service installation and de-installation, billing and accounting, etc. While logically centralised, in practice the control plane will be distributed and replicated for reliability, and only execute on nodes offering a high degree of security.

The nucleus and control plane implementations are "new" to the platform (in the sense that they are written for the platform), but they do not lie on any applications data path or control path. Applications running on more than one node are free to use whatever operating system facilities (e.g. sockets) are available to communicate internally and externally.

Note here that we are *not* advocating the complete absence of APIs in public computing platforms. Such an approach would prevent an application written with the platform in mind from taking full advantage of the platform's facilities, and hence go completely against our philosophy. Rather, we say that any API we provide for applications to use should not syntactically interpose itself between the application and the system. It *is* reasonable to introduce functionality into the system (whether or not by extending the API), but we do not want to place any new abstraction boundary between the application and the OS it runs over. Consequently, we expect applications which are written to be aware of running on the platform to use an API which we provide to request services and information from the nucleus.

We are also not averse to modifying the underlying operating system. Such modifications can be regarded as the extension of the nucleus into the OS kernel and runtime libraries, but note that once again these do not place any new interfaces between the applications and resources.

In terms of commercial offerings, Ensim ServerXchange [6] comes closet to our vision, providing a complete virtual Linux OS to individual service providers. Ensim's offering can be seen as a particular point in the solution space we are addressing in our project: reasonably good resource isolation and an unmodified O/S API are provided at some cost in resource usage (due to the duplication of the O/S) and with no explicit support for availability or heterogeneity of services.

4 CURRENT STATUS

We are in the process of prototyping, deploying and evaluating a public computing platform designed along these lines at Sprint Labs. Our platform consists of 32 rack-mounted dual-processor Pentium III machines connected to two fast Ethernet/IP switches in a mesh. The platform also uses a layer 4/5 switch as a gateway and firewall, and is be connected via dual Gigabit Ethernet links to the Lab, the Internet, and our residential services testbed—200 homes in the town of Pacifica (on the coast just south of San Francisco), each of which has 100Mb/s network access and set-top boxes running Linux.

Our initial implementation uses QLinux [18] as the OS for the platform since it provides the kind of resource control facilities we need, the source code is readily available, and a large number of services already run over Linux.

We intend to implement many services on the platform: if the system lives up to our expectations, deployment of a new service should require little more effort than would installing it by hand on a machine or group of machines, and there are many services out there we can reuse. For evaluation purposes, we have picked three points in the space of service implementations which we feel cover pretty well the range of resource requirements and behaviours we would like the platform to handle:

1. WWW servers: web servers are best-effort applications which nevertheless have scaling requirements. In addition, Web servers typically fork worker processes and CGI programs, and hence provide a good example of a dynamic, multi-process capsule.

2. Streaming media servers: the network and disk bandwidth requirements of streaming media servers provide a useful contrast with the more transaction-oriented and cache-friendly WWW server case.

3. Multi-user games: while game servers do not necessarily have the bandwidth requirements of continuous media applications, delay and jitter bounds on both communication and CPU usage are very tight to ensure interactive response across all users. This provides an instructive additional challenge for the platform.

Our first-draft nucleus implementation is written in C and uses Tcl both as a glue language to abstract from the details of starting, stopping, installing and monitoring the health of each capsule, and also as a way of rapidly prototyping network invocations. Early experience shows that this approach to control mechanisms on a node works well.

Current work includes re-implementing this control structure, by extending an environment like Ninja[7] with native O/S calls and a scripting language. Note that although our principle is to avoid any concept of a Distributed Processing Environment as a basis for the everything on the platform, the case of implementing the control plane is a good example of the limited domain in which DPEs can actually be useful:

a small set of communicating entities which evolve together under a single, tightly controlled architectural authority.

Our initial applications will be unmodified, that is, they will not have been written with our platform in mind. We aim to accommodate a variety of application classes and application-implemented reliability mechanisms. In the later phases of the project we will experiment with adding more knowledge about the platform to applications.

5 OPERATING SYSTEM ISSUES AND RESEARCH DIRECTIONS

Our position on system functionality outlined in this paper might be said to be part of a "pragmatic turn" in systems research, along with systems like Exokernel and Nemesis and away from the excessive definition of abstractions. Rather than being a retrograde step, we feel this opens up fruitful new areas of OS research.

In particular, it is interesting to look at what OS designs and facilities work well with this model of service provision. We feel that from a Quality of Service standpoint, the requirements are similar to ones identified for multimedia operating systems in the early 1990s: resource guarantees, and isolation between application domains to prevent QoS crosstalk. However, appropriate policies and accounting mechanisms may be quite different.

Secondly, the security requirements of Public Computing Platforms are different from more traditional enterprise systems. We expect that OS designs like Eros[17] and KeyKOS[8] will play an increasingly important role in supporting large numbers of mutually antagonistic applications. The extensions of these systems to distributed, cluster-based platforms is in an early stage.

6 ACKNOWLEDGEMENTS

The ideas in this paper have benefitted from many useful discussions with a number of people, including: Supratik Bhattacharyya, Gene Bowen, Michele Clark, Ed Kress, Morley Mao, and JayCee Straley.

REFERENCES

[1] Open distributed processing – reference model: Overview. Recommendation X.901, International Telecommunication Union, March 1997.

[2] A. Acharya, M. Ibel, M. Schmitt, M. Koelsch, D. Magdic, B. Smith, and M. Tuncer. Wrens: A Framework for Rapidly Evolvable Network Services. Technical Report TRCS99-31, Computer Science Department, University of California at Santa Barbara, September 1999.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 45–68, New Orleans, Louisiana, March 1999.

[4] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, Washington, May 1995. IEEE Computer Society.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[6] Ensim Corporation. Ensim ServerXchange. `http://www.ensim.com/products/servercity_fr.html`, November 1999.

[7] S. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The Multispace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, California, June 1999.

[8] N. Hardy. The KeyKOS Architecture. *ACM Operating Systems Review*, 19(4):8–25, October 1985.

[9] Hewlett-Packard. *e"speak Architecture Specification, Version Beta 2.2*, December 1999. Available at: `http://www.e-speak.net/library/pdfs/E-speakArch.pdf`.

[10] J.-F. Huard and A. A. Lazar. A programmable transport architecture with qos guarantees. *IEEE Communications Magazine*, 36(10):54–62, October 1998.

[11] D. Iggulden, O. Rees, and R. van der Linden. Architecture and Frameworks. Technical Report APM.1017.01, ANSA, February 1994. Available from `http://www.ansa.co.uk/`.

[12] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, and R. Fairbairns. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

[13] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. Second Usenix Symposium on Operating System Design and Implementation*, pages 153–168, 1996.

[14] D. Reed. Nemesis: An operating system with principles. `http://nemesis.sourceforge.net/`, February 2000.

[15] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accounted execution of untrusted code. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[16] S. Rooney, J. van der Merwe, S. Crosby, and I. Leslie. The Tempest, a Framework for Safe, Resource Assured, Programmable Networks. *IEEE Communications Magazine*, October 1998.

[17] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island, SC, December 1999.

[18] P. Shenoy. Qlinux home page. `http://www.cs.umass.edu/%7Elass/software/qlinux/`, July 1999.

[19] R. van der Linden. An Overview of ANSA. Architecture Report APM.1000.01, ANSA, July 1993. Available from `http://www.ansa.co.uk/`.