Operating System Requirements for Liquid Software

SRL Technical Report SRL-2000-02

Revision 1.0

Jonathan S. Shapiro, *ed.*
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
shap@eros-os.org

November 26, 2000

Abstract

A new model of software, *liquid software*, is emerging. In this model, software is installed on individual machines, on servers, and within the networking and communication infrastructure dynamically. At one extreme, applications may make changes to the computing fabric very rapidly to support changes in user requirements. At the other, liquid software may be used as a mechanism for upgrades, feature updates, or bug fixes. The liquid model is being pursued in both conventional, real-time, and multimedia applications.

From the perspective of the device builder, a defining characteristic of liquid software is that the platform itself is open. Third parties may install software on your device. These means that application demands on the device cannot be wholly predicted or planned for in advance, which is a particular concern for real-time and multimedia applications.

This document attempts to capture the requirements for such a system. No existing operating system fully meets the requirements for such a system.

Chapter 1

Environmental Assumptions

Our requirements are predicated on a number of assumptions about the applications that will exist in liquid environments and the devices that will run these applications.

A-1. Liquid software systems are characterized by the ability to dynamically install and execute third-party code on behalf of an application without user intervention. By "third party," we mean code that is not written or controlled by the seller of the underlying device.

The security model of operation of a liquid system is to *permit* the promiscuous installation and execution of software (subject to resource limits) and then control what that software does in a flexible but secure fashion. This dynamic installation ability allows systems to be rapidly adaptive to their environment. Eventually, we believe that liquid software come to be a significant feature of future devices.

The ability to *manage* hostile programs has value whether or not dynamic program installation is permitted. An operating system that can manage liquid software must provide finer grain and finer accuracy resource controls than a conventional system.

Note: A liquid software system may *also* provide mechanisms that allow the *operating system* to be updated in the field. This is an orthogonal design issue, and is not a defining characteristic of a liquid software environment. It has impact primarily in imposing the requirement that it be possible to "quiesce" the system in order to allow the operating system upgrade to be performed in an orderly way.

A-2. The applications that run on a liquid device change over the life of the device, and therefore cannot be fully enumerated in advance. Some of the software that runs on a liquid device may be uncooperative or actively hostile (possibly due to bugs). Game software, for example, may use resources aggressively, and this might interfere with video delivery unless properly managed. Software that steals content is an obvious concern.

A-3. Both real-time and isochronous (multimedia) applications must be supported by the a liquid software operating system.

A-4. Consumer devices are part of the liquid device space. An increasing number of consumer devices will have some form of permanent storage (disk drive, FLASH ram, etc.).

A-5. An increasing number of liquid devices will be networked, either within a system of related devices (e.g. home stereo) or in the larger sense of the internet or both. In some cases, such as cell phones, connectivity may be intermittent. In other cases, such as mobile devices, available bandwidth and cost may vary by an order of magnitude depending on environmental conditions (e.g. GSM vs. BlueTooth).

A-6. It is desirable for a uniform programming model to exist across all liquid devices. In particular, devices without permanent storage will often be linked to devices that do have permanent storage. If there is a common programming model, there will be opportunities to reuse software components across these devices.

A-7. Robustness of liquid software is an important concern, and decomposition of applications into independently testable, protected units is therefore desirable.

A-8. A liquid operating system may need to run on a variety of hardware implementations. It should not rely on oddities of a particular processor.

As the discussion in subsequent chapters will hopefully show, these assumptions essentially dictate most aspects of the core operating system design.

Chapter 2

Protection

R-1. Because some software is untrusted (or even hostile) (Assumption 2), isolation must be enforced by the operating system. The mechanisms involved must support isolation, but must also allow controlled interaction between collaborating software agents.

There are two kinds of isolation that need to be considered:

- **Security Isolation.** Broadly, two programs should not be able to interact or use resources unless they are specifically authorized to do so. Requirements for security isolation are addressed in this chapter.
- **Performance Isolation.** Where resources are multiplexed in some way (e.g. real memory is divided among multiple programs), there need to be both upper and lower bounds on how much of these resources programs can use. The upper bound is a limit, and is needed to ensure that *other* programs get the resources they need. The lower bound is a guarantee, and is needed to ensure that a program can obtain the resources required to perform its function. Both bounds must be enforced by the operating system. Performance isolation is primarily addressed by Chapter 3 and Chapter 5, but has implications throughout the system design.

R-2. **Need for Protection.** The only means to acquire authority should be through authorized operations. No application should be able to "make up" authority.

R-3. **Principle of Least Privilege.** No application should have more authority than it requires.

This requirement may be relaxed in the interests of backwards compatibility, but new applications should be restricted to exactly the authority that they need – and no more.

R-4. **Principle of Complete Mediation** Any interaction between two programs, or between a program and an object, should either be intentionally permitted or impossible. That is, all objects and processes should be protected, and use of an object or process should require authorization.

R-5. **Additive Authority** Entities should be created with no initial authority, and any authority they require should then be added. That is, the model should be: "grant the authority that is needed" as opposed to "take away the authority that is *not* needed."

This is a corollary of the principle of least privilege.

R-6. A user (or program) should be able to execute a program with reduced authority. That is, the program runs with less authority than the user has.

This implies that there must be no operation by which the reduced program should be able to gain authority unless this operation has been authorized.

R-7. Processes should have no *implicit* authority.

For example, all POSIX processes implicitly have access to the file system by means of the `open()` call. The `open()` call implicitly references either the *current working directory* or the *file system root* descriptors that are part of the POSIX per-process state. All processes have such descriptors.

While there are cleaner designs for file system interfaces, there is no inherent security problem in allowing some processes to have such descriptors. The fault is in *requiring* that processes have them. Many processes have no need to access the file system at all, and simply should not have such authority. In other cases, under the *Principle of Complete Mediation*, we may wish to interpose control between the process and the file system.

Under the *Principle of Least Privilege, all* authority held by a process should be explicitly and intentionally given to that process.

R-8. **Principle of Explicit Authority** When invoking a service, a program should explicitly identify the authority under which this invocation is permitted.

This is as opposed to having broad authority associated with the process. This requirement ensures that software is written in such a way as to be explicit about the authority it uses and requires.

R-9. Processes should not *inherit* authority on creation.

This follows from the previous requirement. If "inheritance" is desired, the inheritable authorities can be explicitly granted as part of process creation.

R-10. It should be possible to efficiently revoke *all* authoriy to any entity (this is necessary to perform the "destroy" operation on an entity, and also to support pageout).

R-11. It should be possible to selectively revoke a particular principal's authority to any entity.

Note that in a liquid systems, each installer of a piece of software is effectively a principal (or may even be more than one).

R-12. Any security policy that is unenforceable should be ommitted.

R-13. Where possible, the enforceability of security policies should be mathematically verifiable. Too many of them do not work.

This is not to argue that we should spent a lot of time performing rigorous verifications. Rather, this is to insist that we sketch enough of the proof of enforceability that we are confident that a rigorous verification is *possible.*

Chapter 3

Real Time

R-14. **Reserve Requirement** Because some software has real-time requirements, (Assumption 3), it must be possible for applications to "reserve" those resources they require prior to execution, and to revise these reservations during the course of execution.

R-15. **Metadata Reserves** In addition to reserving resources (pages, cpu time), it must be possible for an application to reserve any necessary metadata necessary to *use* those resources.

> COMMENTARY
>
> An example of such metadata is memory mapping data structures. If an application has a reserved page of memory, but is unable to obtain the necessary mapping data structure(s) to map that page, the page cannot be used.
>
> Note that the relationship between metadata usage and data usage depends heavily on application behavior. It is *not* sufficient to consider the allocation of metadata as implicit in the allocation of data.

R-16. **Bounded Operations** Every primitive operation provided by the kernel must complete in tightly boundable time, with low variance.

- The variance should wherever possible be significantly smaller in magnitude than the average time of the operation.
- The variance must not exceed the average time of the operation.

> COMMENTARY
>
> I would prefer to state stronger requirements here, but there are many varying effects in real systems simply due to cache behavior. I am therefore not confident that stricter requirements can actually be met, and I believe that these requirements are sufficient to establish the right "designer behavior" to arrive at a usable real-time system. I would welcome improvements on the statement of this requirement.

R-17. **Bounded Non-Preemption.** In any kernel design there are windows of time during which an operation is non-preemptable. A given implementation shall specify an upper bound on the duration of such non-preemption, and no operation shall exceed this bound.

Note This upper bound probably should not exceed 1ms.

An upper bound is required as a precondition for real-time CPU scheduling, to ensure that preemption variance is bounded.

A corollary of this requirement is that only fully trusted code can be non-preemptible. That is, only the kernel can enter non-preemptible regions.

R-18. Reserve granularity must be per-object.

COMMENTARY

The issue here is achieving an appropriate match between the entities that *consume* reserves and the granularity at which the reserve can be controlled.

For example, a per-process working set reserve in an MPEG player is insufficiently fine grain. File reads that load more movie content use page frames, and other page frames must be freed from the working set to obtain these page frames. As a result, the incoming movie may compete with the movie *player* for residency unless residency for the data and the player can be mutually isolated. The problem with a per-process working set is that it does not isolate them from each other.

This does not mean that every page must be explicitly bound to a particular page frame. Rather, it means that the operating system must support sufficiently fine-grain "scopes" to be established for resource binding decisions.

Chapter 4

IPC

R-19. We wish to support isolation of programs in two ways: security isolation and performance isolation. This requires the introduction of some form of *domain* into the system architecture that identifies the unit of execution to be isolated under some particular policy.

COMMENTARY

The best work on performance isolation to date is probably the work in the Nemesis kernel [LMB+96], and more recently the followup work on performance isolation in RCANE-Menage:RCANE. A forthcoming paper to be published in March provides an extensive discussion of the cost of isolation in language-level implementations, and argues convincingly that operating system-level support is required [AAA+01].

R-20. Domains should be implemented at the smallest possible granularity that is consistent with acceptable performance.

This follows from the *Principle of Least Privilege.*

COMMENTARY

If protection domains were free, then in the ideal world we would point to each instruction and say "this instruction in this execution context needs the following authority," and for each instruction we would build a separate protection domain(probably with compiler assistance).

In practice, this is prohibitive:

- Changing protection domains *does* have cost. Protection domains therefore have performance impact, and engineering tradeoffs must be made between domain-related overhead and satisfying the *Principle of Least Privilege*.
- There is thought and design effort required to determine what the protection domains should contain. If the domain granularity is too small then domains become too difficult to use, and programmers start to become lazy.

R-21. Domains should be implemented at process granularity, not some smaller unit.

COMMENTARY

Experience with HYDRA/C.mmp [WLH81], the Intel I432 [CGJ88], and other hardware and software systems indicates that domains smaller than processes cannot be efficiently supported on modern hardware.

While there has been dissenting views in the literature [CB93], more recent work on L4 [HHL$^+$97] and EROS [SSF99] has shown that process-grain domains can be efficiently supported.

R-22. Since domains are protected, an operating-system primitive is required for performing authorized communications between domains. This primitive is IPC (interprocess communication).

R-23. When an IPC is performed, the authority to perform an IPC to that destination shall be presented as part of the IPC operation.

This follows from the *Principle of Explicit Authority.*

R-24. **No Arbitrary IPC Timeouts** IPC operations should not specify timeouts.

Except where a timeout matches some real timer in a device, there are only two timeout values that are possibly correct: "forever" and "do not wait." When invoking a service, the "do not wait" timeout is in practice not useful, as the service may not be ready to be invoked at the time of the invocation. All other possible values are arbitrary. Either they are excessively long or they prove to be not long enough in some strange case.

When invoking a service, the "do not wait" option creates a covert channel, as it allows the invoker to detect the business of servers.

R-25. **No Server Denial of Service** The IPC specification must allow a service to protect itself from denial of service attacks implemented by callers. In particular, a caller must not be able to indefinitely block a service that is attempting to respond to a service invocation.

R-26. Once initiated, a given IPC operation must complete in a bounded number of cycles without assumptions about the correct behavior of the programs involved. There must be an upper bound on the number of cycles that *any* IPC will take.

That is, even if the recipient or sender misbehaves, the IPC operation must complete in bounded time. For this requirement, signaling failure is considered a form of completion.

Page faults taken on the sender side to prepare the send buffer are considered to be taken before the IPC is initiated.

Blocking while waiting for the recipient to become invokable is considered to occur before the IPC is initiated.

R-27. **IPC Sequenceability.** The IPCs received by a recipient must arrive in some sequential order.

The issue here is that two IPCs must not be allowed to commingle. If two callers invoke a service at the same time, the service should receive the requests in some sequential order, and not some strange mixture of the two requests at once.

R-28. **IPC Sequence Points.** If an IPC can be interrupted or preempted, the design of the IPC subsystem must divide the IPC into atomic units of operations. Interrupts must be permitted only between units of operations, and process states must be introduced that reflect any possible point of interruption/preemption.

That is, an IPC trap operation must be treated like any other instruction. It must consist of well defined sequence points and the consequences of interruption or preemption must be well defined. In particular, the architecture must clearly specify what state is visible to a debugger that halts a process between any two of these units of operations, and what happens to caller and sender if the debugger causes the transfer to be aborted at that point.

R-29. **Atomicity of Authority Transfer.** If the IPC design permits the transfer of authority from one process to another, then all transfer of authority must be performed within a single unit of operation, and this unit of operation must be the final unit of operation of the IPC mechanism.

> COMMENTARY
>
> The issue here is consistency of the access graph. In order for the protection mechanism to be correct, both sender and recipient must agree about which authorities were transferred (and which were not). A complication arises if a sequence point appears after the transfer of an authority in the IPC, because a debugger can terminate the operation prematurely, leaving an inconsistency in the protection graph.
>
> Therefore, we require that no authorities can be transferred until the IPC operation is guaranteed to complete without further interruptions, and that once an authority transfer begins the IPC *must* proceed to completion.

R-30. **Minimization of States.** The IPC design shall require the minimum number of process states consistent with efficiency.

R-31. **IPC Indivisibility.** Every IPC operation should be a single, indivisible operation.

While a multiple sequence point design is feasible, such a design requires more process states than a single sequence point design, and a multiple sequence point design is not required to provide for efficient extended string transfer [SFS96].

R-32. **Bounded Payload.** The payload transmitted by a single IPC operation must be bounded.

This follows from the combination of the *IPC Indivisibility* rule and the *Bounded Operations* rule.

A problem with this is that sometimes a string larger than the bound must be transmitted. A mechanism for two-party extended mutual exclusion should be provided to allow sender and receiver to engage in an extended string transfer protocol without violating the *IPC Sequenceability* rule. This involves a significantly smaller set of process states [either (available, running, waiting) or some closed wait mechanism similar to L4] than an unbounded string mechanism.

R-33. **No Buffering.** The primitive IPC mechanism shall not provide buffering.

Buffering can be provided by user-level code if needed. Experience with Mach [FL94] suggests that nonbuffered designs are both faster and simpler. In particular, they avoid the need for a category of memory management and policy in the kernel.

R-34. **No Page Faults.** In an IPC operation, the recipient cannot be permitted to run a recipient-supplied page fault handler while the IPC transfer is in progress.

A recipient-supplied page fault handler is untrusted, and may never return. If this occurs when a server is replying to a client, the server would be blocked indefinitely, leading to denial of service.

Note that this issue can only arise when the client intentionally tries to compromise a server. It is always possible for a client to guarantee that a valid mapping exists at their own recipient buffer before performing an invocation.

R-35. **Generalized Invocation.** IPCs shall be generalized to invocation.

An IPC can be seen as a special case of descriptor invocation – it is the invocation of a descriptor that names a process. In many cases the kernel will export descriptors to objects implemented directly by the kernel. There is no reason to introduce new system calls to support these descriptors, and many undesirable consequences of doing so.

Therefore, we require that all services, whether implemented by the kernel or by some other processes, should be invoked by the same mechanism, and that the caller process states associated with such invocations shall not be seen to differ in the two cases (the latter requirement is necessary to permit virtualized implementations of kernel services).

Chapter 5

Resource Allocation

R-36. **Specific Accountability.** Every resource shall be allocated from a specific resource pool. Every resource has a current "owning" pool.

R-37. When allocated, the resource shall continue to be associated it's pool. Allocated resources are *not* bound to the allocating process or domain or thread of control.

> COMMENTARY
>
> It is a common design pattern for the operating environment to associate each process with a resource pool and to destroy the pool when the process is destroyed. This is not the *only* useful design pattern (see R-42), and the design of the system should not require this design pattern.

R-38. Resource pools should be suballocatable. That is, it should be possible to create a new resource pool providing some subset of the resources of another resource pool. A mechanism of this kind is required to provide sensible reclamation semantics for shared objects (see R-42).

R-39. **Resource Reclamation.** When a resource pool is destroyed, all resources allocated from that pool shall either be destroyed or shall be returned to some well-defined resource pool.

R-40. **No Kernel Allocation.** Resources shall not be allocated by the kernel.

> COMMENTARY
>
> If the kernel allocates resources, it must have some policy for what to do when the resource pool is exhausted. The problem is that the kernel does not have any understanding of its applications, and is therefore unable *in principle* to make a recovery decision that makes sense at the application level (except, perhaps, by fortuitous accident).
>
> Note that this problem is significantly worsened in component systems, where killing one process can have unforseen consequences for a number of others.
>
> Performing resource allocation at user level (and *not* doing resource allocation in the middle of kernel operations) ensures that the application is always in a position to take action about the absence of resource.
>
> Another reason to avoid kernel allocation is the problem of commingled resources. L4, for example, asks clients to donate pages to the kernel when kernel mapping structures

are exhausted. It then adds these mapping structures to a general pool. The result is that portions of the donated page may be used by multiple processes, and that the first donor effectively subsidizes subsequent clients. Because the resource is suballocated in a way that does not preserve isolation across accountability domains, it becomes impossible for the kernel to return storage to the correct donor.

Fixing this flaw in L4 would require the kernel to introduce a first-class kernel-implemented resource pool abstraction, because the L4 kernel allocates resources during kernel operations, and the pools must therefore be accessable by kernel operations.

Finally, note that even if kernel-implemented pools are provided, there remains the possibility that the pool will be exhausted, and at this point the kernel must do something well defined. In effect, kernel allocation introduces the need for a new kernel abstraction and a new policy, but together these merely serve to push the problem back rather than truly solving it.

R-41. **Lifespan of Resource Pools.** Resource pools shall have a first-class existence whose lifespan is independent of the lifespan of any other resource.

In particular, a resource pool is a distinct entity from a process, and processes do not have resource pools.

COMMENTARY

A resource pool is a first class object that is independent of a process or address space. One common usage of resource pools will be to associate them 1:1 with a process, but this is not the *only* pattern of usage.

If two (or more) processes are sharing some object, the process that created the object should not be required to continue running for the sole purpose of ensuring that the object is not reclaimed. Similarly, it should be possible for one process to give all ownership of an object to a second, including control over when that object is destroyed. This requires that ownership of the resources that the object occupies be transferrable between the processes, and that termination of a process should not implicitly cause the reclamation of the storage pool. The means for performing such ownership transfer is to transfer authority over the resource pool from which the object was allocated.

R-42. **Homogeneity of Allocation.** It must be possible for two processes to share an object in such a way that all of the resources comprising the object come from a single resource pool, and that this resource pool be distinct from the resource pools of either process.

It is *not* required that resource homogeneity be enforced by the operating system. If two processes wish to make "scrambled eggs" they can, and there are sometimes good reasons to do so. The requirement here is that the operating system provide mechanisms that allow processes to build shared objects that have extent and resource allocation clearly distinguished from any member process.

R-43. **Transfer of Resource.** Provided that the process holds appropriate access rights, it should be possible to transfer ownership of resource between two resource pools controlling resources of the same type.

In particular, it should be possible for a process to "donate" some of its resource to another resource pool in order to provide the resource needed by a shared object.

Chapter 6

Notes

An alternative to the L4 flexpage-based mapping design that does not require kernel memory allocation can be found in [JSS00].

Bibliography

[AAA+01] D. Scott Alexander, Kostas G. Anagnostakis, William A. Arbaugh, Angelos D. Keromytis, Paul B. Menage, and Jonathan M. Smith. The price of safety in an active network. *Journal of Communication and Networks*, March 2001.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th Symposium on Operating Systems Principles*, December 1993.

[CGJ88] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance effects of architectural complexity in the intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, August 1988.

[FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating threads model. In *Proc. Winter USENIX Conference*, pages 97–114, January 1994.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian. Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France., October 1997.

[JSS00] *ed.* Jonathan S. Shapiro. First-class flexpage-based address spaces. SRL Technical Report SRL-2000-03, Johns Hopkins University, Department of Computer Science, November 2000. This document captures a collaborative effort by many investigators.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), September 1996.

[SFS96] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. The measured performance of a fast local IPC. In *Proc. 5th International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, USA, November 1996. IEEE.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw Hill, 1981.