Improving Operating System Availability With Dynamic Update

Andrew Baumann
University of New South Wales & National ICT Australia
andrewb@cse.unsw.edu.au

Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski
IBM T. J. Watson Research Center
{jappavoo,dilma,okrieg,bob}@watson.ibm.com

Abstract

Dynamic update is a mechanism that allows software updates and patches to be applied without loss of service or down-time. Dynamic update of an operating system enables administrators to defer rebooting or restarting services and the resultant disruption, without trading off the ability to apply important security fixes or improve functionality and performance through software updates.

We have considered the problem of building a dynamically updatable operating system, and have designed and implemented a prototype update mechanism for the K42 research operating system. Although the prototype utilises the hot-swapping features of K42, many aspects of the design would be relevant for other operating systems.

In this paper we categorise and discuss these issues, and where possible propose solutions. We also describe our current prototype.

1 Introduction

To support on-demand computing, where workloads are unpredictable, high system availability is crucial. Unplanned down-time has long been a problem for computing infrastructure, but in an on-demand environment the cost even of scheduled down-time is increasingly prohibitive. For example, Visa's transaction processing system is routinely updated as many as 20,000 times per year, yet tolerates less than 0.5% down-time [15].

Dynamic update [16] is used to minimise such down-time. It involves the application of software updates to a running system without loss of service. Whereas the normal process for applying an update might be to install an updated version of a program and then restart that program, losing service while the restart takes place, in a dynamically updatable system the new code is loaded into the running program without the need to restart.

A stable and reliable operating system has been important for high-availability computing systems, however modern operating systems are subject to a constant stream of updates and patches. These updates are issued to fix bugs, correct security holes, improve performance, or add features, and they require either restarting system services, or worse, rebooting the entire machine, to take effect. Although these restarts can be planned for and scheduled, it forces administrators to trade off the cost of down-time against the risk of remaining vulnerable to a known security flaw.

A solution to these problems is to make the operating system dynamically updatable, enabling patches to be applied to the running code, and improving the system's availability. We have examined a number of issues involved in designing a dynamically updatable operating system, and implemented a prototype within the K42 operating system.

The remainder of this paper is organised as follows: Section 2 presents and classifies the issues unique to dynamic update in an operating system, Section 3 covers

This work is supported by DARPA under contract NBCH30390004.

our prototype implementation in K42, Section 4 describes the limitations of the prototype and our plans for future work, Section 5 discusses related work, and Section 6 concludes.

2 Designing a dynamically updatable operating system

In order to design an operating system that can be dynamically updated, a number of issues that affect the application of dynamic update techniques to an operating system need to be addressed.

An operating system places special constraints upon a dynamic update implementation, because it plays a crucial role in maintaining the security and integrity of a system, and the update mechanism must be designed with this in mind. Additionally, the basic performance and scalability of the system should not be affected.

To update a system, we must identify the fundamental updatable unit of the system. In an operating system these units may be procedures, modules, subsystems, servers, or some other abstraction. The structure of the system dictates what is feasible, for example in a monolithic kernel it might be sensible to perform an update at the procedure or kernel module boundary, whereas in a microkernel-based system one could choose to dynamically update user-level server processes.

Having identified the updatable unit, a dynamic updating system has to deal with fundamentally the same issues. These are:

1. **Performing the update at a safe point**: in the same way that concurrent systems suffer from race conditions, if a dynamic update is performed in a critical period the system could fail. For example, while the system's state is being modified, an update should not occur. It is therefore important to determine when an update may safely be applied, however since this is, in general, undecidable [10], system support is required. Common solutions involve requiring the system to be programmed with update points, or detecting when the relevant part of the system is idle, or quiescent.

   An operating system is fundamentally event-driven, responding to application requests and hardware events, unlike most applications, which are structured as one or more threads of execution. Because it is event-driven, an operating system often enters quiescent points when no events are being handled, which can be used to avoid relying on pre-programmed update points.

2. **Transferring state information**: unless the unit of the system being updated maintains no state, there must be a mechanism for transferring this state information, so that the updated unit can continue transparently from the unit it replaced. Furthermore, if the replacement unit stores its state in a different structure, there must be a mechanism for transforming the state information to the new structure.

3. **Redirecting invocations**: after the new unit has been installed and the state information transferred, the system must ensure that all future invocations are serviced by the new unit rather than its predecessor.

Limitations of the update system

Ideally any released update could be dynamically applied, that is, any change that can be made to the source code for a system should be supported by our mechanism. However, there are many problems in structuring arbitrary changes as dynamic updates, therefore in practice systems supporting dynamic update have a more restricted definition of what constitutes an update.

Important questions to answer are:

1. **What can be changed by an update?** There is an important trade off between the complexity and flexibility of a dynamic update system, when choosing what kinds of update can be supported. It is simpler to support only changing code, but not data structures. It is also simpler to keep the interfaces to modules fixed, and to only allow changing the code behind an interface.

   These questions directly affect the kind of system changes that can be packaged as a dynamic update, and the complexity of the dynamic update system. For example, if module interfaces can be changed by an update, either the entire update must be applied atomically, or the system must be able to cope with multiple versions of interfaces existing concurrently.

2. **How critical is the timeliness of an update?** For some updates, such as security fixes, it is important to know when an update has completed, and to be able to guarantee that an update will complete within a certain time frame. For other updates, such as performance enhancements that do not affect correctness, timeliness may be less of a concern.

   These differing requirements are one of the key factors to consider in an operating system. It may be that different approaches are required for different updates, in the case of security updates they must be applied either immediately or lazily when the relevant service is accessed, whereas in other situations they could be applied by a background task. Lazy or background application of updates minimises the performance perturbation experienced during update, but raises other issues in change management, and the ability to know what code is actually executing on the machine.

3. **Are updates to middleware or system libraries supported?** Increasingly, important functionality is implemented in middleware and system libraries, both of which are also the target of updates and patches. Dynamically updating these involves performing an update in the address space of each running process, which implies an additional level of complexity between the operating system and the applications. However, without such a feature, updates which change or extend the interface to the operating system are not useful until each application is restarted.

4. **Are updates trustworthy?** Because a misbehaved or malicious update could easily compromise the security and integrity of the system, some mechanism should be used to ensure that only trusted updates can be loaded. Classic approaches here include trusted administrators, and code signing. Alternatively, if the system (and its updates) are coded in a type-safe language, proof-carrying code can be used to verify the safety of updates [12], however re-implementing an operating system in such a language is very difficult.

3 Dynamic update in K42

K42 is an experimental operating system being developed at IBM Research. It is designed to be highly scalable, and features a modular, object-oriented structure, to support rapid prototyping of experimental features (such as dynamic update).

Object orientation is pervasive in K42's implementation—each resource or entity is managed by an object instance [4]. For example, there is an instance of the *Process* object in the kernel for each process in the system (this is analogous to the process control block present in other operating systems). There are presently two implementations of the process object interface, *ProcessReplicated*, the default, and *ProcessShared*, which is optimised for the case when a process is present on a single CPU [1]. A running K42 system could have a combination of replicated and shared process objects present.

To support adaptability, K42 includes hot-swappable objects. Hot-swapping allows an object's implementation to be transparently changed while the system is running. It works by temporarily suspending incoming calls to an object, detecting when the object is quiescent, transferring state to the replacement object, updating a global reference so that future invocations use the new object, and then forwarding the suspended calls to the new object [3]. To date, this mechanism has been used only to support reconfiguration and adaptation on a per-object or per-resource basis [2, 17], however as we will show it can also be used to support dynamic update.

Dynamic update

A good choice for the dynamically updatable unit in K42 is the same as for hot-swapping: the object instance. Hot-swapping transparently changes the implementation of a specific object instance. To extend hot-swapping to dynamic update, the infrastructure must be able to both locate and hot-swap all instances of an object, and direct any new instantiations to the updated object.

To track object instances and control object instantiations required a change in K42's programming model. Previously, object instances were tracked in a class-specific manner, and objects were usually created through calls to statically bound *Create* methods. For example, to create an instance of the *ProcessReplicated* object (the implementation used by default for *Process* objects), the call used was:

```
static SysStatus Create(ProcessRef &,
    HATRef, PMRef, ProcessRef, char *);
```

This leads to problems for dynamic update, because the *Create* call is bound at compile-time, and cannot easily be redirected to an updated implementation of the *ProcessReplicated* object, and also because we rely on the caller of this method to track the newly created instance.

To address these problems factory objects [8] were added to K42. The responsibility for creating and tracking objects is now placed with the factory for that class, and the class has a static member that references the default factory. The majority of these implementation details are hidden behind class inheritance and preprocessor macros. It is worthwhile to note that performance and scalability influenced our implementation of the factories. For example, object instances are tracked for dynamic update in a distributed fashion using per-CPU instance lists. Nevertheless, there is work to be done in benchmarking and optimising our implementation.

We used the factories to implement dynamic update in K42. To perform a dynamic update of a class, the following steps are taken:

1. A factory for the updated class is instantiated.
2. The old factory object is hot-swapped to the new factory object, during this process the new factory receives the list of instances that was being maintained by the old factory.
3. Once the hot-swap has completed, all new object instantiations are being handled by the new updated factory, and therefore go to the updated class.
4. To update the old instances, the new factory traverses the list it received from the old factory, creating a new object instance and performing a hot-swap between the old and the new instances. This step proceeds in parallel across all CPUs where the old factory was in use.
5. Finally, the update is complete and the old factory is destroyed.

We found that adding factories to K42 was a natural extension of the object model, and led to other advantages besides dynamic update. As an example, in order to choose between *ProcessReplicated* and *ProcessShared*, K42 had been using a configuration flag that was consulted by the code that creates process objects to determine which implementation to use. Using the factory model, we could remove this flag and allow the scheme to support an arbitrary number of implementations, by changing the default process factory reference to the appropriate factory object.

Initial experiments

To test and validate our prototype implementation, we constructed an updated version of the *ProcessReplicated* class, named *ProcessReplicatedV2*. This class inherits from, and is functionally equivalent to, *ProcessReplicated*, aside from some minor changes to allow us to detect which version is being used. We then performed a dynamic update, replacing all instances of *ProcessReplicated* with *ProcessReplicatedV2* objects, while the operating system was running its regular set of regression tests, which involves the creation and destruction of a large number of processes.

4 Future work

Our prototype suffers from several limitations. Due to a limitation of the current hot-swapping implementation, and because we only swap a single object at a time, we cannot dynamically apply updates that require changes to object interfaces, nor can we update code that isn't part of a hot-swappable object such as low-level kernel code. We could potentially extend the design of hot-swapping to support changing object interfaces—this would require atomically hot-swapping multiple objects, including the object whose interface is to be changed and all objects possibly using that interface. We have not yet fully considered the requirements of such a feature, nor its ramifications for our quiescence detection algorithm. The severity of the the limitation to only update hot-swappable objects also remains to be seen[1].

[1]We are considering performing an analysis of our revision control system's modification history, to determine the proportion of changes that could have been applied using our dynamic update mechanism.

State transfer between the old and new versions of an object is performed by the hot-swap mechanism using state transfer methods: the old object provides a method to export its state in a standard format, which can be read by the new object's import method. This works well enough, but it requires the tedious implementation of the transfer code, even though most updates only make minor changes, if any, to the instance data (for example, adding a new data member). It should be possible to automate the creation of state transfer methods in such cases, as has been done in other dynamic update systems [12, 14].

In our initial experiment, the code for *ProcessReplicatedV2* was compiled into the kernel ready for use by the update. Clearly this is inadequate for a proper system, however it was sufficient to verify the prototype. We are presently working on a mechanism to load the updated object code into a running kernel or system server, based on a simplified version of the scheme used for loadable modules in the Linux kernel [5].

We need a mechanism to automate the preparation of updates from source code modifications. This could possibly be driven by *make*, using a rebuild of the system and a comparison of changed object files to determine what must be updated. However, it would be extremely difficult, if not impossible, to build a completely generic update preparation tool, because changes to the source code of an operating system can have far-reaching and unpredictable consequences.

Our update system does not yet support updates to system libraries or middleware. At present it is possible to perform an update in an application's address space, however there is no central service to apply an update to all processes which require it. We intend to develop operating system support for dynamically updating libraries in a coordinated fashion.

One largely unsolved problem that must be addressed for dynamic update systems is the problem of configuration management. In order to package and apply an update, or in order to debug or understand the running system, it it necessary to know what code is actually executing. In the presence of dynamic updates, or even with traditional patches, this is not easily determined. Most dynamic update systems that have automated the update preparation and application process assume a linear model of update [12, 14], that is each update depends on all previous updates having been applied before it. In this case it is important to know when an update has completed, and to be able to track which updates have been applied. We will need to consider these issues once we start automating the update preparation process.

We may require a scheme for attaching special-case code to be used in applying certain updates. For example, if an update is designed to correct buggy or misbehaving code, it may first be necessary to forcibly terminate and clean up after the offending code without using the usual update routines. Such a mechanism could also be used to support updates to the update system itself.

Applicability to other operating systems

Our work relies on several features of K42, the object-oriented nature, and the hot-swapping mechanism, however we anticipate that it could be applied to other systems using similar techniques. As long as the system has a modular structure, and uses a unified mechanism for invoking modules (which allows interposition and hot-swapping), it should be possible to add a factory mechanism and perform dynamic updates.

As an example, if adding dynamic update to a system such as Linux, which is structured with kernel modules, one could choose to add a factory-like concept to the module interface, making modules responsible for tracking any "instances" of state that they create. The advantage that K42 offers over mainstream systems such as Linux when it comes to dynamic update is that the system is already decomposed into finer-grained updatable units, and that a larger portion of the system can be updated (in Linux, much core kernel functionality is not modularised).

5 Related work

To our knowledge, no previous work has focused on dynamic update in the context of an operating system. Many systems for dynamic updating have been designed, and a comprehensive overview of the field is given by Segal and Frieder [16]. These existing systems are generally either domain-specific [7, 11], or rely on specialised programming languages [12, 14], making them unsuitable for use in an operating system implemented in C or C++.

Dynamic C++ classes [13] may be applicable to an updatable operating system. In this work, automatically-generated proxy classes are used to allow the update of code in a running system. However, when an update occurs it only affects new object instantiations, there is no support for updating existing object instances, which is important in situations such as security fixes. Our system also updates existing instances, using the hot-swapping mechanism to transfer their data to a new object.

Commercial operating systems commonly offer features such as Solaris' Live Upgrade [18], which allows changes to be made and tested without affecting the running system. However, a reboot is required for any changes to take effect.

Component- and microkernel-based operating systems, where services may be updated and restarted without a reboot, also offer improved availability. However, while a service is being restarted it is unavailable to clients, unlike our system where requests can continue to be handled. Going a step further, DAS [9] supported dynamic update through special kernel primitives, although the kernel was itself not updatable. It remains to be seen precisely which classes of updates can be supported by our system, but as we have demonstrated, there is no restriction on updating the kernel.

Dunagan *et al.* have developed an online analyser [6], which continually traces accesses made by applications to files (and the Windows registry) in order to produce library dependency information. This simplifies the testing of patches, since it is possible to determine in advance which applications might be affected. Online analysis might be useful in our system for determining which objects are affected by an update, however in a development environment where full source code is available, it is most likely cheaper and simpler to track such dependencies statically using the source code.

6 Conclusion

Our goal is to improve the availability of operating systems through dynamic updating. We have presented our prototype of a dynamically updatable operating system based on K42, and discussed the issues encountered in its design that are generally applicable to operating systems.

This paper has raised more questions than it has answered, and there is much research to be done, however we have provided a framework and a prototype for performing that research. We are continuing to develop our implementation, and are using it to explore some of the research issues that we have raised.

Acknowledgements

We wish to thank the other members of the K42 team at IBM Research for their input: Marc Auslander, Maria Butrico, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. We also thank Raymond Fingas, Kevin Hui, and Craig Soules for their contributions to the underlying hot-swapping and interposition mechanisms.

Availability

K42 is released as open source and is available from a public CVS repository, for details refer to the K42 web site: http://www.research.ibm.com/K42/.

References

[1] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. IBM Research Report RC22863, IBM Research, July 2003.

[2] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[3] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma Da Silva, Orran

Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems*, pages 3–8, Charleston, SC, USA, November 2002.

[4] Marc Auslander, Hubertus Franke, Ben Gamsa, Orran Krieger, and Michael Stumm. Customization lite. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, May 1997.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[6] John Dunagan, Roussi Roussev, Brad Daniels, Aaron Johnson, Chad Verbowski, and Yi-Min Wang. Towards a self-managing software patching process using black-box persistent-state manifests. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*, May 2004. Also as Microsoft Research Technical Report MSR-TR-2004-23.

[7] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd ICSE*, pages 470–476, San Francisco, CA, USA, 1976.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. In *Proceedings of the 3rd ICSE*, pages 295–304, Atlanta, GA, USA, 1978.

[10] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[11] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, Annapolis, MD, USA, May 1996. IEEE Computer Society Press.

[12] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–23. ACM, June 2001.

[13] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes—a lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Technical Conference*, pages 65–76, June 1998.

[14] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin, Madison, 1983.

[15] David Pescovitz. Monsters in a box. *Wired*, 8(12):341–347, December 2000.

[16] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, March 1993.

[17] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, pages 141–154, San Antonio, TX, USA, 2003.

[18] Sun Microsystems Inc. *Solaris Live Upgrade 2.0 Guide*, October 2001. Available from http://wwws.sun.com/software/solaris/liveupgrade/.