Dynamic Evolution of Network Management Software by Software Hot-Swapping

N. Feng, G. Ao*, T. White+, B. Pagurek*
Carleton University,
Ottawa, Ontario,Canada K1S 5B6
pagurek@sce.carleton.ca
** Nortel Networks, Ottawa, Ontario*
+Texar Corp. Ottawa, Ontario

Abstract

The computer communications world is very dynamic, requiring continual software updating for correction, perfection, and increased functionality. The problem addressed here is that of providing an evolutionary path for software that permits updating without disrupting the operation and management of the network. This problem is relevant to network management software which is also dynamic. For example SNMPv3 is not yet a standard and is not yet widely deployed. Its initial installations will need to be perfected as more experience is acquired. This paper examines a software hot-swapping solution to the problem, whereby management system software modules can be replaced dynamically without disrupting the management process. The paper also discusses application of the technique to a modular SNMPv3 system implemented in Java.

Keywords: Software hot swapping, SNMP, mobile code, transaction

1. Introduction

Software upgrading, needed for bug fixes, updates, or functionality upgrades, is generally not easy. It is particularly difficult in computer communication networks where software can be widely distributed across heterogeneous domains. The problem is complicated even further by the requirement in some applications for almost 100% availability. As a result, in many cases it is extremely important not to have to take a system off line for software upgrading and/or recompilation. Software hot-swapping means the replacement of a software program or a part of a program while the whole software system remains in operation. Methods that allow for such dynamic software replacement and their application to network management are the object of this research.

Cercio and Pelaggi [2] outline the problem of software management within the framework of network management and the need to adapt it to changing and evolving needs. They also highlight and analyze several types of software maintenance including corrective, adaptive, and expansive maintenance. They

suggest that effective software management allows considerable cost reduction and conclude that expansive maintenance on a system in operation would require an architecture that allows runtime insertion of new modules (without interrupting the service). Our goal is to explore the applicability of hot-swapping as a solution to these maintenance problems.

In the distributed computing world, mobile agent/code environments facilitated by object-oriented techniques are examples of the type of middle-ware that can aid in the construction of dynamic software replacement systems. The SNMP world has also recognized the value of dynamic upgrading and extensibility in recent years. For example protocols like the SNMP Agent-X protocol have been developed to allow the dynamic extension of SNMP MIBs without the necessity of recompiling SNMP agents. The script MIB [3] has been applied to extend the capabilities of remote nodes by delegation. They are steps in the right direction; however, we are searching for a broader solution which, for example, would allow us to install the Agent-X protocol itself in an SNMP agent at runtime, or better yet extend the MIB itself within the agent directly by module replacement. We are concerned with dynamically updating the SNMP management infrastructure itself in a more fundamental way. This is partly a problem of software management but it is also partly a protocol issue.

As a first network management application to consider, we chose upgrading an SNMPv3 entity. SNMPv3 is modular - made up of several subsystems with interfaces that prescribe the interactions between the modules. One of these modules for example is the Message Processing Subsystem that contains in turn a V3MP (Version 3 Message Processing) module as shown in Fig. 4. SNMPv3 is still a not widely used standards track protocol and there remain many outstanding issues of interpretation that are being ironed out and will continue to be resolved for some time to come. Many of these issues will show up as errors in the message processing subsystem making it very useful to be able to upgrade it without having to disruptively take the SNMP entities involved out of service for recompilation and reinstallation. As another example, it might be necessary to fix a bug in the MD5 module of the Security Subsystem, desirably without having to interrupt the service for any consequential interval. It may be desirable in the future to change from the User Based Security Model itself to some yet unknown security model. The change would have to be coordinated with all corresponding SNMP entities employing the new security model, and this could be a formidable task if say hundreds of agents were involved and needed to be recompiled and reloaded. However with some planning ahead the change could certainly be dynamically achieved with considerable benefit through hot-swapping, employing mobile code technology for code delivery and class loading. This is discussed at greater length in section 6. Finally, hot-swapping seems to offer a more fundamental and flexible way of accomplishing expansive additions (new functionality) than attaching them externally to an agent with a new specialized protocol introduced by rebuilding the agent.

To provide a software hot-swap capability it is necessary to provide solutions to determining when it is okay to upgrade, how long should be allowed for the upgrade, what the unit or module for swapping should be and how it can be made swappable, and what the support infrastructure should be. To accomplish the task, one must design an architecture for a swappable module, modularize the application so the

required components are swappable, then design the transactions that will accomplish the goal of getting the replacement code to the desired location, getting the target system into a swappable state, replacing the code, and restart the system with its state restored. All this needs to be done in a fail-safe way while minimizing its side effects on the performance of applications. Whatever the changes to be made, the advantage of dynamic software hot-swapping lies in accomplishing the task with minimal interruption to the system. Anyone who has dealt with large amounts of regular polling data for maintaining an up-to-date feel for the system and for anomaly detection and location, has had to handle missing or "dirty" data when applying his algorithms. This is an impediment to investigating new data-mining and fault management techniques. Being able to evolve the system within a single polling interval is an example of a very desirable goal.

This paper continues with a description of a number of terms and a description of the issues in object swapping. The paper then discusses the software hot-swap architecture, its components, and the hot swap mechanism in some detail. Subsequently, we explain how this architecture has been used to dynamically upgrade SNMPv3 entities and the issues arising.

2. Software Hot-Swapping

Before we describe our software hot-swap architecture in detail, we first examine some of the swap-related problems.

2.1 Issues in Object Swapping

First, there is the *Referential Transparency Problem*. In Figure 2a, object A asks object B for service. If we want to use object B1 to replace object B, object A has to change the reference from B to B1. This means that object A must know the change. However, as depicted in Figure 2b, if A has passed B's handle to object C, then C will retain the reference to B even if A has changed the reference to B1. Thus, if we want to swap object B with B1, we have to make all the objects that get B's handle replace the reference to B with the one to B1. Here, the replacement of an object is not

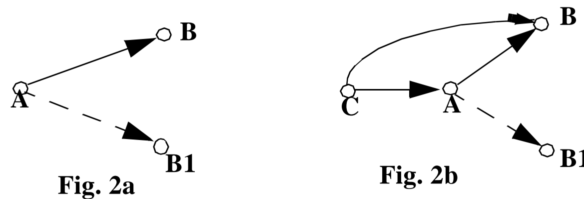

Figure 2. Referential Transparency Problem

transparent to its internal clients. In some cases, the object to be swapped may have no idea about its internal clients, and thus have difficulty in notifying its clients of the replacement.

Second, we have the *State Transfer Problem*. To achieve the objectives of hot-swapping, we have to transfer the state of object B to B1 so that the application runs consistently. The state of an object summarizes its history, namely its attribute values and its current execution status. However, if the object B is executing a number of

operations concurrently (if it is multi-threaded), its state may keep changing. This makes the transfer of state from one object to another a complex task.

Third, we have the *Mutual Referential Problem*. This problem severely complicates hot-swapping, as modules may depend upon each other for services. In such cases, multiple modules must be swapped in one transaction and the order of swapping may be important, possibly because of the State Transfer Problem outlined in the previous paragraph. At a higher level the SNMP manager(client)-agent(server) relationship falls within a similar category if an upgrade changes the protocol and necessitates synchronized swap transactions on both sides of the relationship.

2.2 Requirements

It must be pointed out that software hot-swapping should not affect the robustness of the S-Application. In other words, if the swap involves swapping multiple interacting S-modules, even at remotely located nodes, in order to maintain the integrity of the system the swap might have to be atomic or all-or-nothing. All related swap transactions need to be successful or the operation fails. This is similar to the nature of SNMP's atomic set operation when multiple variables are to be set. If the hot swap transaction fails, the S-Application should be able to continue to provide services to its clients. This requires that the swap transaction should be a two-phase commit transaction, which means that the transaction should be either committed or aborted and rolled back.

The swap transaction should also complete in a reasonable amount of time. What is timely depends very much on the application. High availability systems such as those that occur in telecom switching allow for very small downtime at any time. Network management, while not considered critical real-time work can be compromised if critical information like a notification is lost or there are excessive delays in processing. This can easily happen in large networks simply because of the many players involved, and in fact, this is where speed in hot-swapping becomes critical.

Finally, we require that all S-Applications be written in Java in order to take advantage of its location independence, dynamic configurable class loaders, reflection, and garbage collection facilities. There is however a price to pay in slower execution and problems with different JVM versions. Java also produces some obstacles by hiding part of a module's state in the JVM. We recognize that much of the environment could be accomplished in C++ but only with a great deal of extra work to simulate Java facilities like reflection.

2.3 Research on Software Hot-Swapping

Useful technology for hot-swapping is limited in availability and capability. The state of the art up to 1993 was well reviewed in [6] where it was pointed out that dynamic linking was not the best technology to solve the problem and that dynamic updating cannot be done if some kind of indirection between the program modules that invoke each other cannot be incorporated into a language or its underlying runtime system. Moreover techniques for preserving the correctness of a program being updated must also be provided. A telecom application of dynamic software replacement [7]

involved use of the specialized Chorus real-time operating system for swapping at the process level. It was limited to process-structured programs using only message-based communication between the processes. Typically reported replacement times were of the order of 20 to 50 milliseconds depending on what was being swapped and its size.

More recently the rapid maturity of Java and recent work on mobile agents and code [9], [14] have contributed to available technology building blocks by making code delivery to an active application fairly straightforward. Probably the closest work to ours is contained in the Dynamic Component Updating System [11] that uses Java and a wrapper architecture to provide the necessary indirection. Their system like most such systems provides for transfer of only part of the modules state and so is limited to swapping non-active objects. We are the first team to publish on software hot-swapping in network management with a design based on mobile code and Java.

In our research, we analyzed several approaches [1]. These were a Java Virtual Machine (JVM) modification approach, an Observer Pattern approach, a Proxy Pattern approach, and a Mediator Pattern approach. The last three approaches are based on design patterns [5]. A design pattern addresses a recurring design problem that arises in specific situations and describes a solution. Modifying the JVM would make manipulation of object references possible thus solving the referential transparency problem. It would also make available the system stack etc. thus making the entire state of a module accessible. Such a technique was used in the NOMADS system [13] to facilitate what is termed strong mobility for mobile agents. This feature would allow the entire state of a mobile agent to be captured so the agent may safely resume operation at a remote site from exactly where it left off before moving. This ability would be extremely useful for hot-swapping because it might allow a swap to occur at any instant. After a comparative analysis of advantages and disadvantages, described in some detail in [1], the Proxy Pattern approach was implemented to provide the indirection needed to solve the referential problems. JVM modification is held in reserve for systems where very high availability is critical enough to warrant employing a non-standard JVM to solve difficult state transfer problems.

3. New Software Hot-Swapping Infrastructure

The following sections provide a detailed design and rules for designing an S-Module.

3.1 Architectural Elements and Services

In our architecture, shown in Figure 1, an S-Application is composed of S-Components each made up of an S-Module and an S-proxy, Non-S-Module components, and a Swap Manager. As described in detail in [1], only an S-Module can be swapped. In order to control an S-Module swap transaction we introduce the Swap Manager. An S-Module can cooperate with other S-Modules and non-S-Module components that form part of the same application program. Together, they provide services to clients at run-time, and communicate with the Swap Manager. Clients in our system may be external, demanding service from the S-Application, or

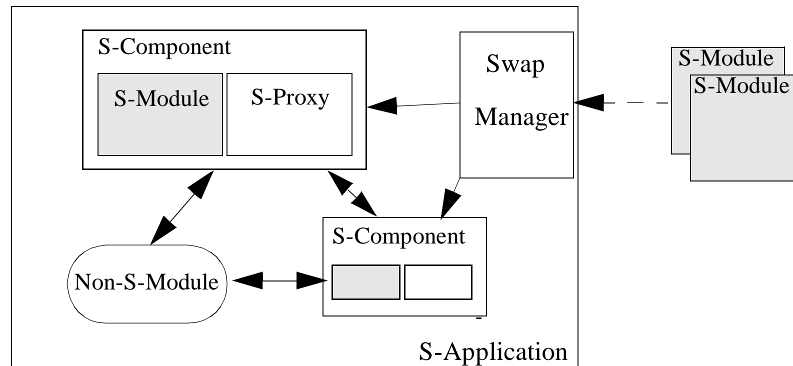

Figure 1. Software Hot-Swapping Architecture

internal, meaning the interaction of one S-Application module with another. The Swap Manager has access to all S-Modules and provides the services shown in Table 1:

TABLE 1. Swap Manager services

Service	**Explanation**
Listening	Waits for new S-Modules and instantiates them
Security	Performs authentication of incoming S-Modules
Transaction	Provides control of hot swapping transaction
Timing	Ensures timely completion of swap transaction
Event	Notification of hot swap events to observers
Repository	Caches S-Module states during transaction

The biggest difference between an S-Module and a non-S-Module is that the upgrading of an S-Module will not interrupt the running of the S-Application, while upgrading a non-S-Module generally needs re-compilation, re-linking and re-starting the S-Application. When we design an S-Module, special characteristics have to be taken into consideration, such as attribute mapping and interfaces.

3.2 Characteristics of an S-Module

- **Identity**: An S-module has a unique identity that is made up of S-Module type and S-Module identifier.
- **Version**: An S-Module has an associated version with major and minor indices and annotation that describes the nature of the changes from a previous version.
- **Service**: The same type of S-Modules should have the same set of services. For example, a security S-Module should provide the following basic services: encryption, decryption, authorization, and authentication. As long as two S-Modules have the same set of services, they can be swappable.

- **Finite State**: An S-Module should have a finite set of internal states. In some states an S-Module is swappable, in others not. If and only if an S-Module is in a swappable state, can the hot-swapping transaction take place. Any service provided by the S-Module should finish within a finite time, and the S-Module returns to an “idle” state after completing the service.
- **Dependency List**: Any S-Module should have a dependency list that includes the S-Modules that it depends on. Before an S-Module can be activated, its dependent S-Modules must have been loaded into the S-Application.
- **Mapping Rules**: Associated with each S-Module is a set of mapping rules, which control the transfer of its internal state between versions of the S-Module.
- **Persistence**: During a hot-swap transaction, the system resources held by an S-Module, such as opened files, in-service communication channels, etc., should be released or transferred to the new S-Module.

3.3 S-Component Design using the Proxy Pattern

The Proxy Pattern can be used to provide access control to an object by having a surrogate or placeholder for it [5]. Since this is the approach we adopted, we discuss it in some detail. Each S-Module has one proxy object (here we call it the S-Proxy object) associated with it. This is shown in Figure 3. An S-Module and its S-Proxy constitute an S-Component. Using the Proxy Pattern, the hot-swapping system contains one Swap Manager, several S-Components and other non-S-Module objects. The S-Proxy is *not* swappable, it is created once when the S-Application starts up and the corresponding S-Module object is first instantiated.

A hot swapping scenario in the Proxy Pattern approach looks like this:

1. The Swap Manager gets a message from its listening service. The incoming message includes all the information about a new S-Module, including its identity, the version of the S-Module, and the code for instantiating objects, etc.
2. After the security service finishes the authentication successfully, the incoming new S-Module is instantiated.
3. The Swap Manager searches for the S-Proxy of the current S-Module (here we call it the old S-Module) with the specific identity, and sends a message to the S-Proxy to block any new method calls, and starts the timing service.
4. Then the Swap Manager checks with the S-Proxy if the old S-Module is in a swappable state. If it is not in a swappable state, the Swap Manager will wait until all the interactions with this S-Module finish.
5. When the old S-Module enters the swappable state, the Swap Manager calls the S-Proxy to get the internal state of the old S-Module and map it to the state in the new S-Module. The mapping rules are applied here.
6. After the new S-Module has been successfully initialized, the Swap Manager initiates a hot-swapping transaction by changing the reference to the old S-Module in the S-Proxy with the reference to the new S-Module.
7. The timing service stops timing and checks to see if the transaction is finished. If yes, this hot-swapping transaction is successful. The Swap Manager releases the blocked calls. The old S-Module can now be removed.

8. If during the step 4 to step 6, the timing service gets a time-out event, this transaction fails. The Swap Manager sends out a notification about this transaction and releases the blocked calls.

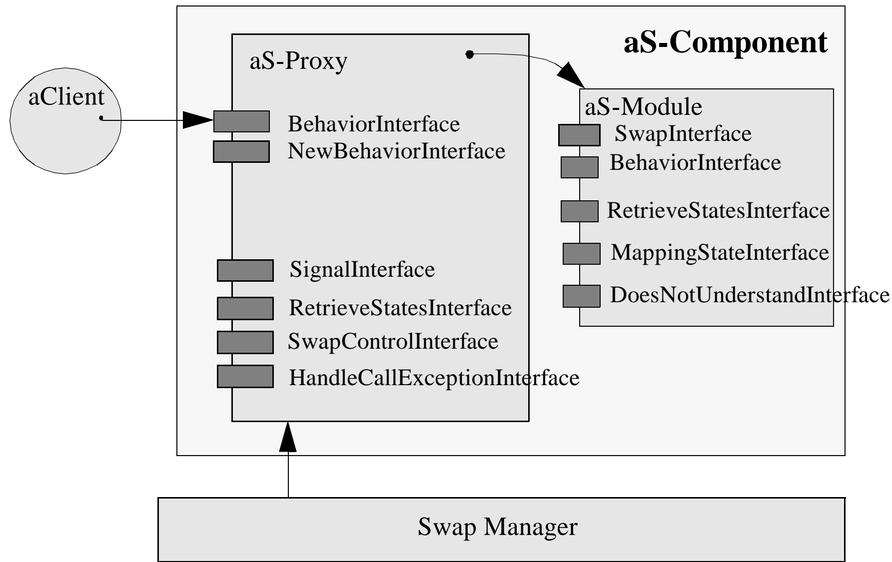

Figure 3. The S-Module and its S-Proxy

3.4 S-Module Design Discussion

The Proxy Pattern approach solves the “Referential Transparency Problem” described in Section 2.1, because all the client objects only have references to the S-Proxy. Only the S-Proxy has a reference to the S-Module. In other words, the client objects and the S-Module are decoupled by the S-Proxy. As a result of this the client objects need not be aware of the swapping transactions between the S-Proxy and the Swap Manager.

This approach is more suitable for situations where the interfaces of the S-Proxy do not change, although some of the methods of the old and the new S-Module can be different. This is so called dynamic messaging which will be discussed in the next section. A general solution to the problem in which the S-Proxy’s interface changes is the subject of ongoing research.

Each S-Module object has one S-Proxy object created for it when the S-Application starts. The hot-swapping transaction will not generate new S-Proxy objects. If the system has many S-Modules, the overhead of creating and managing the same number of S-Proxy objects cannot be ignored.

As a result of the requirement for dynamic messaging [8], Java reflection is used. Although Java reflection allows for the identification of all the members that are associated with an object (field members and method members) and makes it possible for that object to interact with them, using it can affect system performance significantly. There is again a trade-off between the flexibility of the architecture and

its performance. For network management, the flexibility in extending functionality likely outweighs the performance issues.

4. S-Module and S-Proxy Interface Design

This section specifies the S-Component.

4.1 S-Module and S-Proxy interfaces

As shown in Figure 3, an S-Module must implement the following interfaces among which only the BehaviorInterface corresponds to the services provided by the originating module:

- **SwapInterface**: This interface allows the S-Module to participate in a swap transaction; preparation for swapping, voting and cleanup upon transaction commit.
- **BehaviorInterface**: This interface shows the actual behavior of an S-Module and allows its S-Proxy to use services provided by this S-Module.
- **MappingStateInterface**: This interface maps the states of the same kind of S-Modules from one version to another. The Swap Manager uses this interface to initialize a new S-Module properly.
- **RetrieveStateInterface**: This interface gets the states of the current S-Module, or it could return the states of any of the previous versions.
- **DoesNotUnderstandInterface**: This interface handles the exceptions that a service is requested but it is not provided by this S-Module.

Also, the corresponding S-Proxy must implement the following interfaces:

- **BehaviorInterface**: This interface publishes the behavior and services of the underlying S-Module; the implementation here is to forward the external calls to the corresponding S-Module.
- **NewBehaviorInterface**: This interface is used by the external clients to send dynamic messages to the methods that do not exist in the S-Proxy's Behavior-Interface.
- **SignalInterface**: This interface is used by the Swap Manager to check and control access to the S-Proxy.
- **RetrieveStatesInterface**: This interface is used by the Swap Manager to retrieve the states of the current corresponding S-Modules during a hot-swapping transaction. The request is passed through to the associated S-Module.
- **HandleCallExceptionInterface**: This interface allows the designer of an S-Component to handle the exceptional cases that may happen inside the S-Proxy.
- **SwapControlInterface**: The SwapControlInterface extends the SwapInterface. Through this interface the Swap Manager controls the transaction of the hot-swapping procedure. Security is applied here to ensure that only the Swap Manager uses the interface.

4.2 Client interaction scenarios after hot-swapping

Once an S-Module has been hot-swapped, clients begin to interact with it. Three interaction scenarios are possible of which the last two are not mutually exclusive.

New and old S-Modules have same BehaviorInterface

This is a simple case since the new S-Module and the old S-Module have the exactly same behavior methods. The activity inside the S-Proxy will be the same as with both the old and the new S-Module. In this case a client (aClient) sends a message to the S-Proxy (aS-Proxy) by calling the sameMethod with parameter args. Then aS-Proxy forwards the call to the just swapped-in S-Module (newS-Module) directly. The NoSuchMethodException will not occur because the newS-Module has the same method as published on the aS-Proxy.

New S-Module does not have all old S-Module methods

In this case, the new version S-Module has changed the behavior methods in its BehaviorInterface. This could happen when the new version of an S-Module stops supporting some of the older version's functionalities, or simply because the new S-Module requires different parameters for a particular method. Here, one of the behavior methods in aS-Proxy is the oldMethod; aClient sends a message to aS-Proxy by calling the oldMethod with args as the parameter. Since the newS-Module does not support this oldMethod, a NoSuchMethodException will occur in aS-Proxy while it tries to forward the call to the newS-Module. This NoSuchMethodException will be caught and handled by the HandleCallExceptionInterface of the S-Proxy. Finally, aS-proxy will send a message to the newS-Module's DoesNotUnderstandInterface with the method name (oldMethod) and the parameter (args). From now on it is the newS-Module's responsibility to diagnose and make further decisions.

New S-Module has methods that old S-Module does not

In this case, the new S-Module has added some new methods in its BehaviorInterface. This is more likely to happen when the caller is also an S-Module that has been hot-swapped. The client S-Module (here we call it clientS-Module) needs to call the new S-Module's new method (aNewMethod). However, the S-Proxy of the new S-Module does not have this aNewMethod in its BehaviorInterface. Here the following steps need to be taken:

1. The clientS-Module must send a message to the S-Proxy's NewMethodInterface indicating the name of the new method in which it is interested (aNewMethod) and the necessary parameters (args).
2. The S-Proxy will check if the new S-Module has the required new method. Here, Java reflection will be used.
3. If yes, the new S-Module has that new method with the parameter, the S-Proxy will invoke the new method.
4. If no, a message will be sent to the new S-Module's DoesNotUnderstandInterface. It is again the new S-Module's responsibility to make further decisions.

5. Transactions

There are two types of transactions in our hot-swap architecture. The first one, called an S-Application transaction -- which is invoked by an S-Application's clients -- is for the application to provide service to its clients. The second one -- called a swap

transaction -- constitutes a two-phase commit protocol that is used for hot-swapping S-modules and is invoked by the Swap Manager. The protocol used for hot-swapping is inspired by the Jini transaction protocol [10].

5.1 The S-Application Transaction

Our hot-swap technique was applied in a distributed environment. The remote clients may invoke an S-Application transaction through a socket with a proprietary application-layer protocol, or standard distributed computing technologies such as HTTP, Java RMI, CORBA, etc.

In a distributed computing environment, it is natural that clients may initiate several transactions concurrently, especially through RMI or CORBA. Therefore, our hot-swap technique supports concurrent S-Application transactions. Otherwise, it will be very limited from a practical point of view. However, supporting concurrent transactions is a very challenging proposition for a hot-swap technique. An S-module may execute several operations simultaneously; consequently, it is quite difficult to get a persistent state from an old S-module and transfer it to the new one. Modifying the JVM would alleviate this problem but in our SNMP agent implementation this was not considered necessary.

Any S-module to be swapped should agree to being swapped before the swap transaction begins. To do this, the Swap Manager should block new calls to the S-module and allow it to complete its existing operations. Again, if the instantaneous state could be captured then it might not be necessary to allow the old S-module to complete all the operations in progress and reach an idle state.

As mentioned previously, the swap transaction is invoked by the Swap Manager in order to swap the old S-module out of the application and make the new one operational. To ensure the consistency of the application, the swap transaction should be either committed or aborted. Naturally, if several new S-modules have certain inter-dependency relationships, either all or none of them will be swapped in the same swap transaction. Because of space limitations, details of the swap transaction and the complete state machine are left to [1]. We simply reiterate that with the S-proxy approach, it is necessary to queue incoming requests while allowing the involved module to reach a swappable state, and then commit or abort as needed.

6. Hot-Swapping within the SNMP Context

The UQAM Java implementation of SNMPv3 [15] was chosen because of its very modular implementation making it easy to identify several modules as candidates for hot-swapping. As indicated in section 1, the entire Message Processing Subsystem and the V3MP component were obvious candidates, as were the User Based Security Model itself and the MD5, SHA, and DES modules.

Figure 4 shows the S-module decomposition implemented in our first experiments with an SNMPv3 agent. The larger S-Modules are shown shaded and some of the smaller S-Modules and their corresponding S-Proxies are shown unshaded. Provided that the replacement S-module is purely corrective or perhaps adaptive, but does not change the protocol, and provided that the operation can be accomplished fast enough, then the swap can be carried out at only the target node,

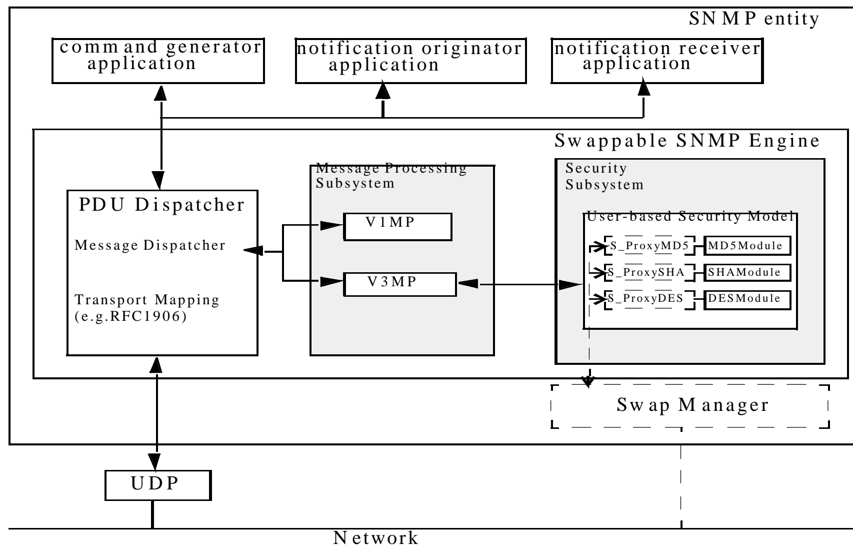

Figure 4. SNMP Entity S-Modules

independent of its remote client-server partner. Using estimated times from [4], if the SNMP messages are small, for example, packets of the order of 1000 bytes, for an agent to finish processing its current packet and to reach a swappable state, should not take more than approximately 50 milliseconds. If very large datagrams are being used for efficiency, it can take longer to reach the required state. We have measured swap times - the time the S-module is actually out of service - of under 30 milliseconds (comparable to times indicated in [7]). The time depends more on the complexity of the inter-module dependencies and the state to be transferred than on the size of the modules. This measurement indicates that there should be no difficulty in slipping a swap transaction into any reasonable polling interval unnoticed, unless the agent is severely overloaded for other reasons. If USM authentication is used, and if network latency of up to 500 milliseconds is assumed, there is very small risk of authentication failure due to delay because USM allows delays of up to 150 seconds. Even with very large SNMP requests, there should be no trouble finishing off processing the current packet and carrying out the swap transaction.

Now consider a situation in which the replacement module contains some kind of protocol change that requires synchronization between the manager and agents, for example, if the MD5 algorithm were being replaced by some new SuperMd12 algorithm, or the encryption algorithm was being upgraded. This situation means that swap transactions at both ends would need to be synchronized by a "global" swap manager which would have to insure system integrity with a multi-phase commit procedure, but at this higher level. If only a single manager and a few agents were involved, there still would not be a problem. If, however, 1000 agents were involved and an atomic global swap were required, then all the local swap transactions must

succeed at all the heterogeneous SNMP entities involved or the whole process must be rolled back. Just processing acknowledgments alone from 1000 remote agents can take 5 to 10 seconds. Because bottlenecks tend to occur at the manager, clearly a two phase commit procedure over a WAN would have to be done with care and efficiency. The speed and robustness of hot-swap transactions would be needed in this management situation and also should make the update process feasible.

First of all, the replacement S-module needs to be sent out from the managing station to all the participants, and, to insure synchronization the code needs to be delivered and ready for swapping before the local swap transactions are initiated. Although we want the swap to occur quickly once it has started, it usually does not have to be done immediately; as a result, there is usually time to prepare. There are several ways this process can be carried out both within and outside of the SNMP protocol. Within SNMP the script MIB [3] provides for a manager pushing scripts to another SNMP entity. With the appropriate instrumentation, Java programs could be included. Outside of the SNMP protocol, there are several mobile code dispatching technologies that can be included, like RMI and even CORBA. In any of these cases, it would be useful to multicast the dispatching process or have it carried out from a separate code server so that the load on the manager station can be minimized. We must remember that the SNMP manager is still carrying out its regular polling duties, polling each agent every 5 minutes, for example, and the manager and network may already be under a severe load.

Assuming that the replacement S-module has already been marshaled at the remote nodes, it is now time to initiate the synchronized swap activity. This could be done with a multicast command to proceed. Network latency should not be an issue. Otherwise, if the remote clocks were reasonably synchronized, the Schedule MIB associated with the Script MIB could be employed to initiate swap execution at essentially the same time. Normally, the two phase commit approach requires that each participant signal that it is prepared to do the swap. However, there should be no problem in omitting this signalling, if the normal SNMP requests are not individually too heavy, and if suitable rollback provisions have been made (the normal SNMP set request does not require pre-acknowledgment).

7. Conclusions

This paper has described an architecture and implementation for the swapping of modules within a running application with minimal disruption of service provided by that application. In our research, we have found that the Proxy Pattern approach seems more reasonable than others; however, this conclusion is based on the assumption that the S-Proxy is not swappable. Based on our measured performance, we believe that the approach appears quite feasible for upgrading SNMP during regular operation. We have not yet applied it to a large scale situation with many nodes, but preliminary analysis is encouraging. Moreover, if faster swap transactions are required, we are prepared to go the modified JVM route, in which instantaneous state capture is possible. The result would be that the time to reach a swappable state once the swap transaction has been initiated, would be greatly reduced. While the research reported here has focused on an important network management software entity -- an SNMPv3 entity -- the work is more widely applicable within the network

management domain, which includes service management. Our current work involves automating the generation of S-modules and proxies, exploring the applicability of global safe checkpointing to the general problem and determining just what is swappable (the concept of an S-critical section has arisen in this context).

8. Acknowledgments

We would like to acknowledge the financial support provided by Nortel Networks We would also like to thank Professor Omar Cherakaoui of UQAM, the University of Quebec in Montreal for the Java source to their Modular SNMPv3 agent [15].

9. References

[1] Feng, N., Ao, G., White,T., and Pagurek, B., "Software Hot-swapping Technology Design", Technical Report SCE-99-04, Systems and Computer Engineering, Carleton University, Ottawa, Canada, June 1999.

[2] Cerchio, L., and Pelaggi, A., "Software Management: An Important Side Issue of Network Management", IEEE NOMS '92, pp.383-396.

[3] Levi, D., and Schonwalder, J., "Definitions of Managed Objects for the Delegation of Management Scripts", IETF RFC-2592, May 1999.

[4] Stallings,W., "SNMP, SNMPv2, SNMPv3 and Rmon 1 and 2", 3rd Edition, Addison Wesley Publishing Co., 1999.

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns, Elements of Reusable object-Oriented Software", Addison-Wesley Publishing Co., 1995.

[6] Segal M., and Frieder, O., "On-the-fly Program Modification: Systems for Dynamic Upgrading", IEEE Software, March 1993, pp.53-65.

[7] Hauptmann, S., and Wasel, J., "On-line Maintenance with On-the-fly Software Replacement", Third International Conference on Configurable Distributed Systems, IEEE Computer Society Press, pp 70-80, 96, May 1996

[8] Blewitt, A., "Using dynamic messaging in Java", available at URL: http://www.javaworld.com/javaworld/javatips/jw-javatip71.html.

[9] Nelson, J., "Programming Mobile Objects with Java", John Wiley & Sons, 1999.

[10] Waldo J. "Jini Architectural Overview" available at URL: http://java.sun.com/jini/whitepapers/.

[11] Plasil, F., Balek, D., and Janecek R., "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", ICCDS '98, International Conference on Configurable Distributed Systems, Annapolis, May 4, 1998.

[12] Laddaga R., and Veitch, J., "Dynamic object technology", Communications of the ACM, Vol. 40, No. 5, pp. 36-38, 1997.

[13] Suri, N., Bradshaw J., et al "NOMADS: Toward a Strong and Safe Mobile Agent System", Agents 2000, The Fourth International Conference on Autonomous Agents, Barcelona, June 2000.

[14] Bieszczad, A., White T., and Pagurek, B., "Mobile Agents for Network Management", IEEE Communications Surveys, 1(1) : 2-9, 1998.

[15] "Modular SNMP," available at URL: http://www.teleinfo.uqam.ca/snmp/.

[16] "Mobile Code Toolkit," available at URL: http://www.sce.carleton.ca/netmanage/mctoolkit/