Accelerating Code Deployment on Active Networks

Tôru Egashira and Yoshiaki Kiriha

C&C Media Research Laboratories, NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki 216-8555, Japan
Tel: +81-44-856-2314, Fax: +81-44-856-2229
{egashira,kiriha}@ccm.CL.nec.co.jp

Abstract. Active networks enable their users to specify how each packet is processed on network nodes. One of the essential techniques for active networks is the programmable node approach, which enables network nodes to evolve their packet processing functions by loading new software components into the nodes. Possible component loading strategies include demand loading. It reduces the usage of node resources and localizes possible problems, although the component loading time defers packet processing. This paper discusses the component loading time of the demand loading strategy, and proposes a scheme to shorten the loading time by masking the propagation delay of components. We have implemented the scheme *pre-supplying* on a Java-based system and have evaluated its effectiveness. The result shows that the proposed scheme shortens the loading time of a test component by as much as 70%.

Keywords. Management of active networks, Mobile code, Demand loading, Code server, Prefetching

1. Introduction

As mobile code technology including mobile agents and applets has been improved and widely accepted, applying it to the underlying network system has been explored. Active networks [1] are ideal examples as they stand on mobile code technology. The use of mobile code in active networks is split into two approaches: *programmable node* and *capsule*. The former enables network nodes to evolve their functions including routing, modifying and forwarding of the received packets by loading new software components into the nodes. The latter approach embeds a piece of program code in each packet, or "capsule" in this context, so that the behavior of each packet on nodes is self-described. These approaches complement each other; the programmable node approach can define a large set of packet handling functions that cannot fit into a single capsule, while capsules can define the handling of individual packets by choosing which handling functions to use and making minor additions to them.

Since a capsule may decide its next hop during its journey, predetermining which nodes should have software components for the capsule is

R. Stadler and B. Stiller (Eds.): DSOM'99, LNCS 1700, pp. 211-223, 1999.

difficult. Therefore, the possible deployment strategies of software components are limited as follows:

- Broadcasting, i.e. distributing the components to *all* nodes in advance
- Accompanying, i.e. letting the components accompany the capsule
- Demand loading, i.e. nodes load the components when the capsule arrives

None of these alone are suitable for every purpose because each of them has its own advantages and disadvantages. The demand loading strategy, on which this paper focuses, limits the deployment of components to where they are needed, thus reducing node resource usage and localizing possible problems, e.g. the emergence of undetected bugs and compatibility problems among the new and existing components. However, the packet processing must be deferred during the component loading time.

This paper discusses the component loading time of the demand loading strategy, and proposes a scheme to shorten the loading time by masking the propagation delay of components. We have implemented the scheme *pre-supplying* on a Java-based system and have evaluated its effectiveness.

2. System Model

Before discussing loading time, we define a simplified model of a demand loading system. Our system model includes three types of entities: *module*, *supplier*, and *consumer* (Figure 1).

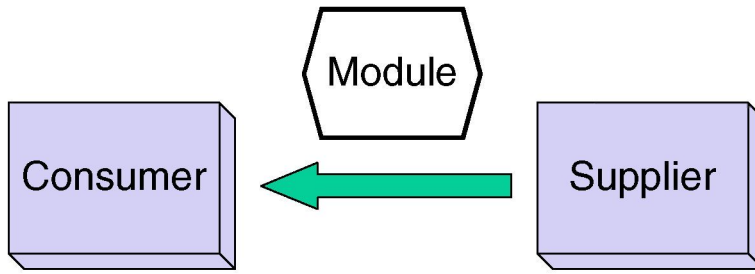


Figure 1: System model.

2.1. Module

A module is the unit of a software program. All software components are composed of modules. Modules are atomic; i.e. they have no substructures. In other words, no module contains any other modules. It is also the unit of program transmission; components are transmitted from a supplier to a consumer module-by-module.

Programs can be split into two types: binary codes and human-readable, also known as scripts. A module may be either type and the difference is observed only as initialization overhead. Each module must be initialized

in order to be ready to run. The initialization includes the compilation and linkage processes. For example, a script typically needs to be compiled to an intermediate language or a binary code before running. Compared with scripts, binary codes generally need less initialization overhead. We assume the initialization of a module cannot be done concurrently with its execution. Consequently, initialization must be finished before the module is used.

2.2. Supplier

A supplier is a subsystem that provides consumers with modules. A supplier is assumed to store all modules that consumers potentially need. Each module that a supplier stores has a unique name. A consumer requests a supplier for a module by its name. The supplier then transmits the module specified to the consumer.

2.3. Consumer

A consumer is a subsystem that executes components that are composed of modules. It may execute many components concurrently. It requests a supplier to send modules that are required to continue the component execution. Upon receiving a module, a consumer compiles it, links it, and/or does on it any other required initialization tasks to make it as the part of the running component. Some of the modules that compose a component may be left unrequested if the application quits the execution prematurely.

2.4. Example

An example that conforms to our system model is the Java applet system. In this example, a Java class file is a module. Java class files compose applets, which correspond to components. An HTTP server and a web browser correspond to a supplier and a consumer respectively.

3. Module Granularity

In this section, we consider the effect of module granularity on the component loading time. In this paper, the granularity of a module is defined by its size in bytes. Moreover, we define component loading time as the elapsed time between the date a consumer starts loading a component and the date the component becomes ready.

An example of the effect of the module granularity can be observed on Java applet systems that support packaged class files. A large Java applet which is composed of a number of class files takes considerable loading time for a browser to start running the applet if each class file is loaded one-by-one via an HTTP connection. Therefore, newer browsers have the capability to load packaged class files. With packaged class files, a browser can obtain all class files in a package via a single HTTP connection. As the number of request-response roundtrips becomes smaller, the browser can save the applet loading time. Such loading time reduction can be regarded

as the result of module granularity tuning.

Concerning the elements of component loading time, they are (Figure 2):

- the module loading time,
- the module initialization time, and
- the component execution time elapsing until the component becomes ready.

The module loading time is further divided into:

- the propagation delay, and
- the module transmission time.

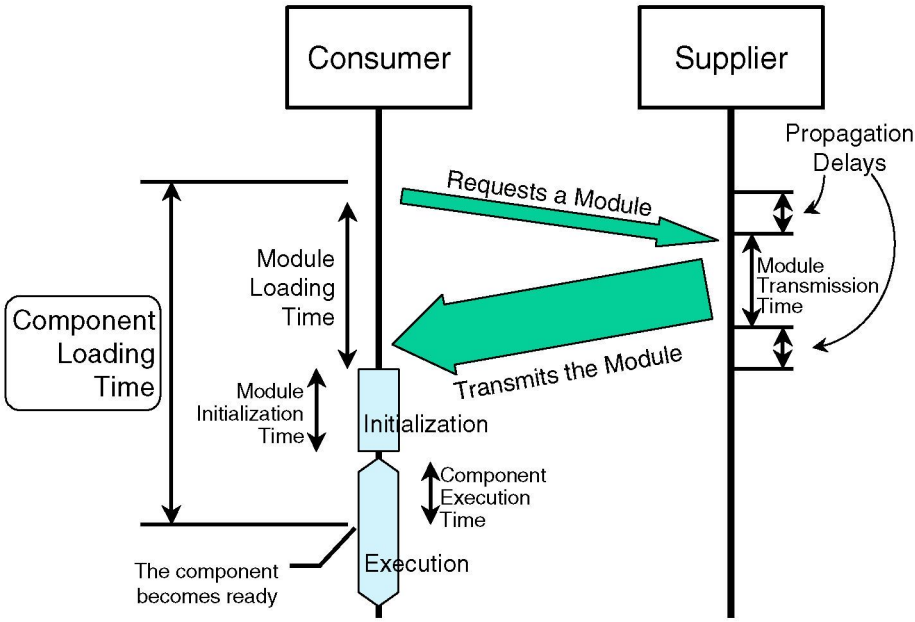


Figure 2: Component loading time.

The granularity of a module affects its initialization time and transmission time. Hence, making the module granularity finer will reduce the component loading time (a to b in Figure 3). However, making the module finer than a certain amount will increase the component loading time (c in Figure 3). As an excessively small module cannot carry enough quantity of code, the consumer will have to load another supplemental module, but this will add another propagation delay to the component loading time. It thus follows that to minimize the component loading time, the module granularity chosen must be as small as possible but still enough for a component to become ready by loading a single module. This means that a "kick-start" module must be prepared for each component.

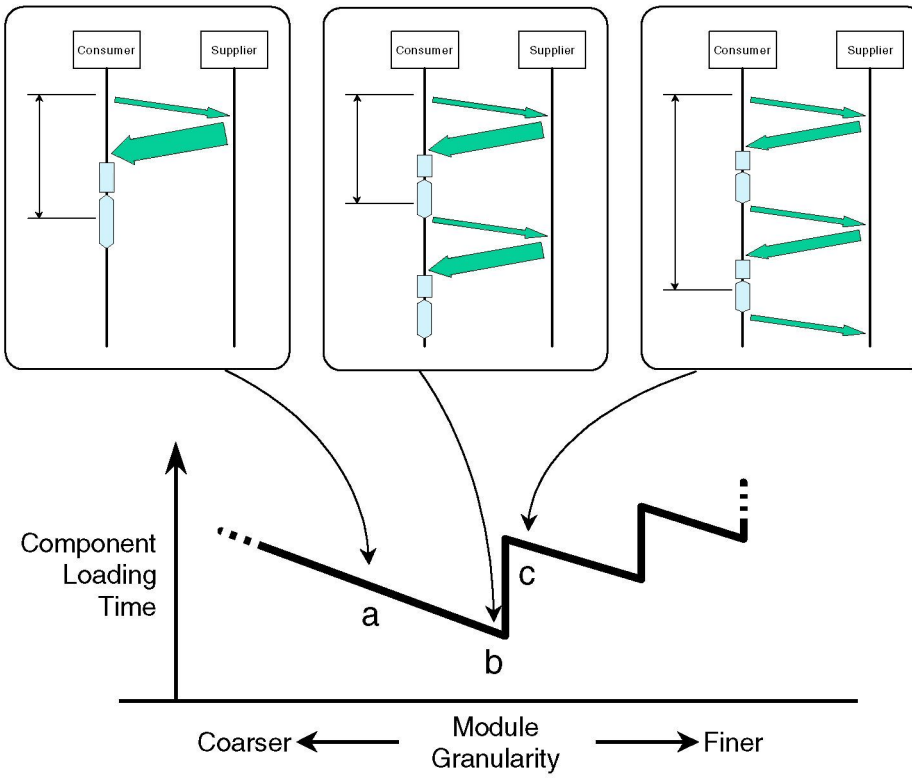


Figure 3: The influence of module granularity on component loading time.

However, as such tailored modules cannot be shared among components, more resources and longer loading time may be required than in finer module approaches when the consumer executes many components. Since an active network node is considered to execute many components, we sought another approach.

4. Masking of Propagation Delay

As described above, propagation delay accumulation is the main problem of modules with excessively fine granularity. However, if the propagation delay can be properly masked, using finer modules will result in shorter component loading time. A technique that can mask the propagation delay is *prefetching*. It is widely used in the memory systems of computers. In this technique, a memory system attempts to guess which data are likely to be accessed by a processor. Then it moves the data from the high-latency storage into the low-latency cache before the processor actually accesses them. In the rest of this section, we discuss existing prefetching techniques and propose an improved technique.

4.1. Prefetching Techniques

Different prefetching techniques are applied to memory systems that are accessed in different ways. Sequential prefetching, in which the memory system prefetches successive data, is suitable for disk cache systems and the paging system for virtual memory implementations because they store data of various length into successive fixed-size memory blocks and thus successive blocks will likely be accessed one after another. In the opposite case, where data are rarely accessed sequentially, sequential prefetching is not suitable. A data cache subsystem for a processor is an example; although the instruction cache is accessed in a nearly sequential manner, the data cache is accessed sparsely. In cache systems for filesystems and the world wide web (WWW), a datum is specified with the name rather than a number, so there is no *a priori* idea of "successive data" and sequential prefetching is not suitable. For such systems, hints and history are used to complement the "successive data" information.

A hint is a piece of information that specifies when and which data should be prefetched. Based on obtained hints, a prefetcher fetches and caches the specified data. A kind of such hinted prefetching technique is adopted in processors. These implement hints as extra instructions inserted in software. Then they notify the address and the length of data to be prefetched to the memory system. Such a scheme is also called "software prefetching" [2]. In the same manner, the use of `ioctl` system calls to declare the future access to a file is proposed for filesystems [3]. Though the hints sources in these examples are the consumers (consumer-hinted), the suppliers of data or other entities can also be the source. Such an example will be described later.

Here, "history" refers to the existing information about past accesses. By analyzing the history, a prefetcher can discover access patterns, extrapolate them, and then predict which data will likely be accessed. As the implementation of this history scheme is more complicated than the schemes mentioned previously, it is typically applied to software-based systems like disk caches [4], filesystems [5], and WWW [6].

Among these prefetching schemes, hinted prefetching and history-based prefetching can be applied to mask propagation delays of module loading. This paper focuses on the history-based prefetching, and leaves the consumer-hinted prefetching for future study.

4.2. Issues of History-Based Prefetching

There are other history-based prefetching issues beyond the implementation complexity described above. For example, since the prefetcher must have sufficient history to predict correctly, it cannot work well for a certain period after being initialized.

A solution to this issue is introduced in a WWW caching system [6], in

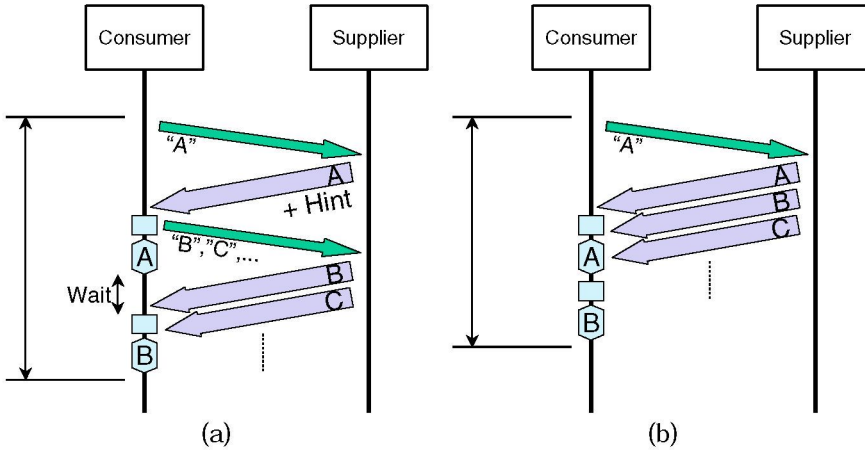


Figure 4: Example sequences of (a) hinted-prefetching and (b) pre-supplying.

which the history is centrally maintained on the server side. The prefetcher in the client uses hints that the server provides based on the history. Hence, the system uses both hinted and history-based prefetching. As the history is maintained by the server, the prefetcher located in the client side can work well from the beginning, as long as some other clients have representatively built up enough history by previously accessing the server.

This solution, however, creates another issue. Because of the propagation delay of hints, it may fail to mask the propagation delay of prefetched data. Figure 4a describes an example in which we have applied the history-based hinted prefetching solution to our system model. This example assumes the supplier has access history of modules A, B, and C in that order. Therefore, when the consumer requests module A, the supplier transmits a hint, which tells him that modules B and C will be next. Upon receiving this advice, the consumer requests modules B and C. Though the second request is made quickly, the consumer has to wait for some period since necessary modules are locking at this point. Thus, component loading time is lengthened. This issue cannot be solved with the prefetching or "consumer-driven module providing" approaches because they incur hint propagation delay.

4.3. Pre-Supplying

We resolve this issue by making the module providing supplier-driven; it is mostly the same as history-based hinted prefetching, except that the supplier transmits modules instead of hints to the consumer. In the case of the last example, the supplier transmits module A on receiving the request for it. Then it further transmits modules B and C (Figure 4b). Thus, the

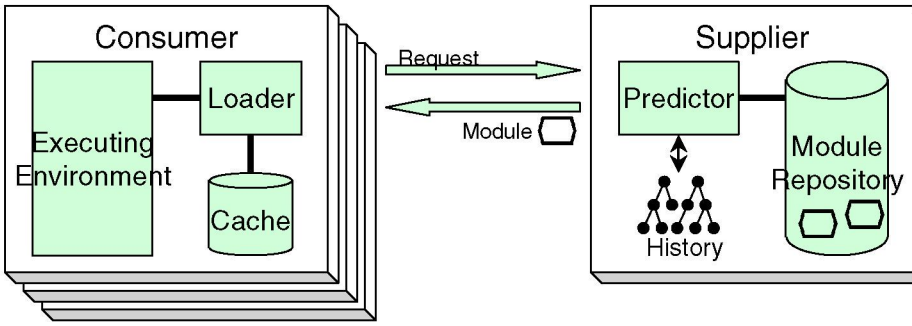


Figure 5: The architecture of pre-supplying system.

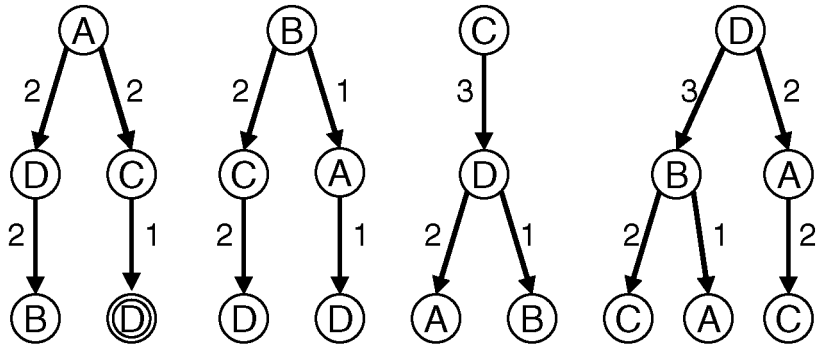
consumer does not have to wait for module B, or the wait time is minimized. To make this scheme called "pre-supplying" feasible, a consumer must always be ready to receive modules, regardless of whether it has requested them.

5. Architecture of Pre-supplying System

In this section, we describe the architecture of the pre-supplying system. The supplier contains two subsystems: a module repository and a predictor (Figure 5). The module repository is a subsystem that provides the predictor with modules on request. The predictor sends modules to consumers on request and on expectation. When it receives a request for a module from a consumer, it loads the requested module from the module repository, transmits it to the consumer, and updates the history data. Additionally, the predictor analyses the history data to predict which module is most likely to be requested next, and transmits the candidate module to the consumer if the module has not been transmitted to the consumer recently. The predictor then repeats the pre-supplying task by transmitting a module that is most likely to be requested after the module that it just pre-supplied. The task is further repeated until the next request from the consumer arrives or when there are no more unsent candidates left.

Figure 6a shows a possible representation of history data. It is a forest of height limited to $h=2$ and thus can model h order Markov processes. Each root node of the forest corresponds to a module, and the subsidiary nodes are modules requested subsequent to the request for the root module. Each edge has weight that is the frequency of the request sequence. For example, when the predictor has observed the sequence of requests shown in Figure 6b, the forest of Figure 6a is made, and it then predicts that module D is the candidate for the next transmission.

Let us return to Figure 5. The consumer contains three subsystems: an executing environment (EE), a loader and a cache. The EE executes



(a) An example of history represented as a forest.

A⇨D⇨B⇨C⇨D⇨A⇨C⇨D⇨B⇨A⇨D⇨B⇨C⇨D⇨A⇨C

(b) The observed sequence of requests.

Figure 6: An example of history data.

software components. While executing components, it loads necessary modules from the loader. The loader provides the EE with modules on request. When the EE makes a request for a module, the loader first looks for the module in the cache, which is a cache for modules and is assumed to be empty initially. If the cache already has the requested module, the loader gives it to the EE. Otherwise, the loader makes a request for the module to the supplier. Some time later, it receives the requested module and some pre-supplied modules. It stores them in the cache, and if they include modules for which the EE is asking, it gives the modules to the EE.

6. Experiment

We conducted a series of experiments to show the efficiency of the pre-supplying. Figure 7 shows the outline of the test bed system. All the software was developed using Java: the supplier and the consumer were developed as Java applications, and each was deployed on a Java Runtime Environment (JRE) version 1.1.7 virtual machine on a Linux PC. The consumer and the supplier are interconnected with a TCP/IP socket on 10 Mbps switched Ethernet. Using the socket, the consumer requests modules (Java class files) and the supplier transmits them.

We evaluated the loading time of a Java application. The application we used is an MPEG filter, which is an experimental code for programmable nodes in active networks. As shown in Figure 8, it forwards an MPEG stream in RFC 2250 [7] Section 3 format. When the output link is congested and the queue is going to be filled, it selectively drops packets in less-impact-first manner (B pictures first, next P pictures, and finally I

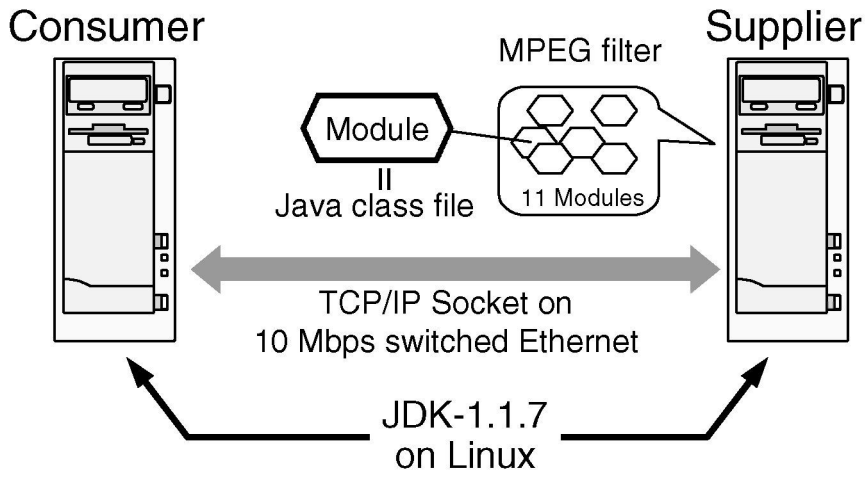


Figure 7: The test bed system.

pictures). The loading time is defined as the time between the date the consumer starts loading the application and the date the application indicates it accepts MPEG packets. During the loading time, the supplier transmits 11 modules, the average size of which was about 608 bytes.

We executed the system with pre-supplying (after history data are established) and without pre-supplying (history data are initialized), 20 times for each. The results are shown in Table 1. The results obtained show that pre-supplying reduced the loading time by 74% approximately.

Table 1: Loading time comparison.

	Loading time [sec]	
	Average	Std. deviation σ
Without pre-supplying	0.536	0.042
Pre-supplying	0.138	0.004

The result in Table 1 was the ideal one because it was only a single application that established history and was executed, hence no prediction errors were made. We also evaluated how the performance is degraded according to the prediction errors. There are two types of prediction errors: ordering and missing errors. Suppose that a consumer is going to use modules A, B and C in that order; if the supplier predicts that the order

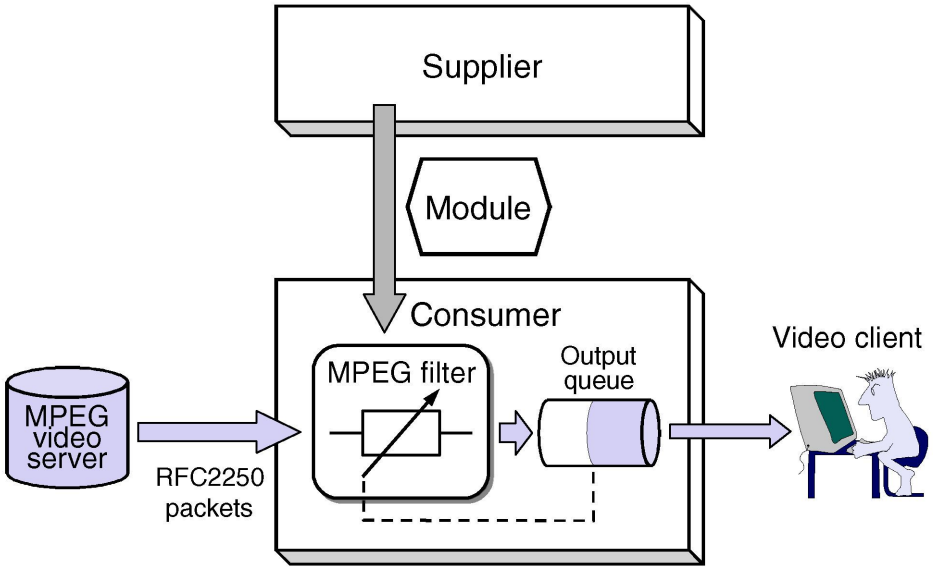


Figure 8: The experimental active networking system.

will be A-C-B, it is an ordering error; if the supplier predicts A-C but no B, it is a missing error. In some cases, an ordering error might not be a problem. For example, pre-supplying in order A-C-B will be sufficient for the consumer that uses them each a day. In contrast, a missing error always leads the consumer to make a request for the missed module (provided that the consumer cache did not have the module yet) and thus lengthens the component loading time.

We evaluated the effect of only missing errors because the effect of ordering errors is less directly observed as described above. To see the effect of missing errors, we forced the supplier to make mistakes. We modified it so that it could omit some randomly chosen modules from pre-supplying. Although it predicts correctly and tries to pre-supply candidate modules to the consumer, some of them are not really transmitted. Because the number of modules in the MPEG filter is not enough for this experiment, we used W3C's Jigsaw Proxy Package[†] 2.0.1 instead. In this case, the supplier transmits 216 modules during the loading time; the average size of modules was about 2.80 KB. As shown in Figure 9, the pre-supplying performance dropped linearly from maximum 68% loading-time reduction according to the number of missing errors; i.e. the number of omitted modules.

[†] http://www.w3.org/Jigsaw/

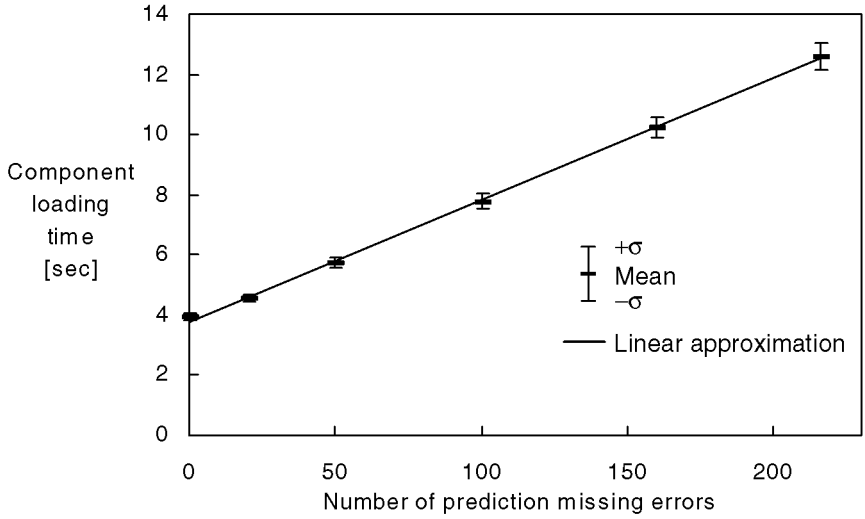


Figure 9: Performance degradation due to prediction missing errors.

7. Conclusion

In this paper, we have presented a scheme called "pre-supplying" to reduce the loading time of software components for active network nodes. In this scheme, the supplier maintains access history, predicts future accesses, and transmits modules in advance of the requests for them. A series of experiments shows that it reduces the component loading time by approximately 70% and that the reduction will linearly decrease according to the number of prediction errors.

On the test bed system, the information exchanged between the consumer and the supplier was limited to the module requests and the modules themselves. We believe more information including cache usage must be exchanged to make this scheme practical, otherwise the supplier cannot know how many modules can be safely pre-supplied without flooding the consumer's cache.

References

1. D. L. Tennenhouse, et al., "A Survey of Active Network Research," *IEEE Communications Magazine*, Vol. 35, No. 1, pp. 80-86, January 1997.
2. S. P. VanderWiel and D. J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *IEEE Computer*, Vol. 30, No. 7, pp. 23-30, July 1997.
3. R. H. Patterson et al., "Informed Prefetching and Caching," *Proc. 15th*

ACM Symp. Operating Systems Principles, December 3-6, 1995, pp. 79-95.
4. K. Salem, "Adaptive Prefetching for Disk Buffers," NASA Goddard Space Flight Center, CESDIS TR-91-46, January 1991.
5. J. Griffioen and R. Appleton, "The Design, Implementation, and Evaluation of a Predictive Caching File System," Technical Report CS264-96, Univ. Kentucky, June 1996.
6. V. N. Padmanabhan and J. C. Mogul, "Using Predictive Prefetching to Improve World Wide Web Latency," *ACM SIGCOMM Computer Communication Review*, Vol. 26, No. 3, July 1996.
7. D. Hoffman, et al., "RTP Payload Format for MPEG1/MPEG2 Video," RFC 2250, January 1998.