More Efficient Network Class Loading through Bundling

David Hovemeyer and William Pugh
Department of Computer Science
University of Maryland
daveho@cs.umd.edu, pugh@cs.umd.edu

Abstract

In this paper, we describe *bundling*, a technique for the transfer of files over a network. The goal of bundling is to group together files that tend to be needed in the same program execution and that are loaded close together. We describe an algorithm for dividing a collection of files into bundles based on profiles of file-loading behavior.

Our motivation for bundling is to improve the performance of network class loading in Java. We examine other network class loading mechanisms and discuss their performance tradeoffs. Bundling is able to combine the strengths of both on-demand strategies and archive-based strategies. We present experimental results that show how bundling can perform well in a variety of network conditions.

Introduction

Through *class loaders*, the Java[1] virtual machine (JVM) supports a flexible model for loading executable code and resources at runtime [8]. *Network class loading* is an important form of class loading, in which class files and resources are transferred to the client JVM over a network from a server.

Because networks are generally slower in bandwidth and latency than local disks, users can experience poor startup times and significant delays when running an application loaded from a network. It is therefore desirable to optimize the transfer of classes and resources over the network to minimize these delays. Specifically, we would like to find an approach that has the following properties:

- Only files needed by the client are sent over the network.
- Files arrive when needed, in the order they are needed.
- As few bytes as possible are transferred.
- The number of requests made to the server should be minimized, in order to reduce the delay due to network latency.

In addition, we would like to make the approach practical for 'real-world' deployment:

- The server implementation should scale to a large number of clients.
- The client code should be small and efficient, and should use only standard parts of the Java runtime (i.e., no native code).
- The client code should have minimal overhead.

This paper describes a technique called *bundling* that can approximate all of these properties under realistic network conditions.

The document is structured as follows. Section 1 reviews existing network class loading mechanisms. Section 2 describes bundling. Section 3 presents some experimental results comparing the effectiveness of bundling to existing network class loading mechanisms. Section 4 describes related work. Finally, section 5 summarizes the contributions of this paper and suggests possibilities for future work.

1 Overview of mechanisms for network class loading

This section discusses issues encountered in network class loading, and how these issues are handled by existing technologies for network class loading in Java.

1.1 Issues in network class loading

We identified several design issues that must be addressed in any network class loading technique. This section discusses these design issues and some of the tradeoffs they imply.

One network connection *vs.* **multiple.** When the client requests class files and resources from the server, it could use a separate network connection for each file, it could use a single persistent network connection for all files, or it could use a fixed number of persistent connections. Using a single persistent connection is generally the best approach. Using separate connections for each file is generally not a good idea since setting up a network connection is expensive. It is not clear that there would be any advantage to using multiple persistent connections, since in class loading there is no obvious advantage to downloading files in parallel.

Transfer granularity: individual files *vs.* **many.** The ability to send individual files is desirable because it allows the files sent by the server to exactly match those needed by the client. As the granularity of the transfer units increases, the probability that unneeded files might be sent increases, as does the probability that files will be sent in an order different from the request order. However, small transfer granularity increases the likelihood of of delays due to network latency, because the client must send requests to the server more frequently.

Using compression *vs.* **not.** Compressing class files and resources is desirable when network bandwidth is limited, since it results in transferring fewer bytes over the network. However, the benefit of transferring fewer bytes must be weighed against the runtime cost of performing the compression and decompression.

Compressing individual files *vs.* **multiple.** Applying compression to multiple files is desirable since it offers more opportunities for sharing redundant information between files, thus increasing the compression ratio.

Performing compression on-line *vs.* **off-line.** Performing compression off-line is desirable because it reduces the amount of computation required on the server.

Pre-sending/prefetching *vs.* **not.** When classes and resources are transferred on-demand, the client must send requests for the files it needs to the server. If files are requested only at the precise point they are needed, then client must pay the cost of network latency for each request. This latency cost can be reduced or eliminated if files are either pre-sent or prefetched prior to the point when they are needed by the client. However, care must be taken that files are not pre-sent or prefetched unnecessarily.

1.2 Existing mechanisms for network class loading

This section describes existing mechanisms for network class loading, and how they address the issues described in section 1.1.

1.2.1 Downloading individual files

One of the standard network class loading mechanisms in Java is for the client to request files individually relative to a directory URL, typically using the HTTP protocol.

Downloading individual files has the advantage that only those files explicitly requested by the client are transferred over the network. This ensures that no unneeded files are transferred, and that the files arrive in the order in which they are needed. HTTP connections are usually persistent, meaning that a single connection is used to transfer many files, and that the connection persists between client requests. In principle, it is also possible for the HTTP client and server to negotiate the use of compression; however, such compression is per-file and does not share redundant information between files. (In any case, the JDK 1.2.2 implementation of HTTP does not support such compression.)

Downloading individual files has the disadvantage that files are not pre-sent or prefetched, meaning that the network latency cost is paid for each file requested by the client. Given that network latency can be hundreds or even thousands of milliseconds, this disadvantage can be considerable.

1.2.2 Downloading Jar archives

Another standard network class loading mechanism in Java is to download Jar archive files [10]. Jar archives use the same underlying format as Zip files [5]. Each file in the archive is usually (though not always) compressed using the Deflate algorithm [4].

Jar files are convenient for application developers because they provide a simple mechanism for packaging an entire application or a component of an application into a single file. Since Jar files contain multiple files, they can be considered a form of pre-sending by the server, meaning that the network latency cost of the request is paid only once, for the entire Jar file, rather than once for each class or resource file. Jar files also offer random access to the files contained in the archive, which is useful for loading classes and resources from a Jar archive on disk. However, when used to transport classes and resources over a network they are likely to deliver unneeded files or to deliver files in the wrong order, because they typically contain a large number of files. In addition, the fact that the files are compressed individually results in a lower compression ratio than would be possible if the entire archive were compressed using cumulative compression.

JDK version 1.3 introduced *Jar indexes*. A Jar index is a mapping from class and resource files to the names of the Jar files in which they are located. The Jar index for a set of Jar files is contained in a *master Jar file*. Once the master Jar file and the Jar index are received by the JVM, subsequent Jar files are downloaded only if they contain a class or resource file needed by the running application. This feature is an improvement over the technique used by earlier JDK versions to find class and resource files in a collection of Jar files: each Jar file was simply downloaded in sequence until the desired file was located. Through the use of a Jar index, an application can be split up into discrete components which are downloaded only when needed. This can help reduce the number of unneeded classes and resources downloaded. (The functionality of the Jar index is in some ways similar to our concept of *bundling*, which will be described in section 2.)

1.2.3 On-the-fly compression

On-the-fly compression is a variation on downloading individual files. Files are requested individually by the client, and transferred from the server over a compressed stream. Any suitable compression algorithm could be used; zlib [1] is an obvious choice since it is a standard part of the Java API (in the `java.util.zip` package)[2].

On-the-fly compression has the advantages of downloading individual files: only the files requested by the client are transferred, and the files arrive in the order needed. The fact that the entire stream of files is compressed cumulatively means that a high degree of sharing of redundant information between files is possible, resulting in compression ratios higher than those achieved by simply compressing each file individually.

However, on-the-fly compression has the disadvantage that it can require considerable CPU time on the server to perform the compression. For example, a 333 MHz Sun Ultra 5 can compress about 2.5 megabytes per second using zlib at the lowest compression level, and about 0.5 megabytes per second at the highest compression level. Considering that server will have more work to do at runtime in addition to compression, the sustainable throughput will be somewhat less. This limits the scalability of on-the-fly compression using zlib.

Another disadvantage of on-the-fly compression is that it does not use pre-sending or prefetching, meaning that the network latency cost is paid for each file requested.

1.2.4 Pack

Pack is a custom archive format for Java class files, developed by William Pugh [9]. It exploits regularities in the Java class file format to achieve a high degree of compression, and is designed as an alternative to Jar files. Like Jar files, the Pack format has the advantage that a large number of files can be delivered in response to a single client request, meaning that the request latency cost is paid only once for the entire archive, rather than per-file. It shares the disadvantage that unneeded files may be sent, and that the files may not arrive in the correct order.

Pack must be downloaded prior to use. The decompressor requires 36 kilobytes when downloaded as a Jar file.

Pack's decompressor is slower than zlib's. On a 333 MHz Sun Ultra 5 workstation it can decompress about 75-120 kilobytes per second, limiting Pack's effectiveness for fast networks.

1.3 Comparison of existing network class loading mechanisms

Table 1 summarizes existing network class loading mechanisms in terms of how they address the design issues discussed in section 1.1, along with how those issues would be addressed by an 'ideal' network class loading mechanism. None of the existing mechanisms has all of the properties we would like in an ideal mechanism. The request granularity of the archive formats (Jar, Pack) is larger than we would like. On-the-fly compression has the desired request granularity, but requires the compression to be performed at runtime, and does not use pre-sending or prefetching to reduce latency costs.

Jar indexes offer an intriguing possibility: we could break the application into chunks, put each chunk into a separate Jar file, and use the Jar index to inform the JVM which class and resource files are contained in each chunk. This scheme has some advantages over using a single monolithic Jar archive. If we choose the division of files into chunks carefully, so that only files needed together are put in the same chunk, then we can avoid downloading files that are not needed. We can also try to order the files within the chunks such that they match the client's request order at runtime. However, this scheme has some undesirable properties. If we want to allow a truly arbitrary mapping of files to chunks, the size of the Jar index will be proportional to the number of files. Also, we are still left with the problem that the chunks are encoded as Jar files, so the files within the chunks are compressed individually, not cumulatively.

	number of net connections	request granularity	compress?	compress scope	compress time	pre-send/ prefetch?
‘Ideal’	1	individual file	yes	all files	off-line	yes
Individual files	1[3]	individual file	maybe[4]	individual file	off-line?	no
Jar archive	1	archive[5]	usually	individual file	off-line	yes
On-the-fly	1	individual file	yes	all files	runtime	no
Pack	1	archive	yes	all files	off-line	yes

Table 1: Summary of existing network class loading mechanisms and an ‘ideal’ network class loading mechanism.

2 Bundling: a hybrid approach

Bundling is an approach to transferring files over a network that tries to combine the benefits of individual file downloading, Jar file downloading, and on-the-fly compression. The collection of files comprising the application is divided into groups, or *bundles*. Each bundle is then compressed cumulatively using either a general-purpose compression mechanism such as zlib, or a Java-specific archive format such as Pack.

As in individual file downloading, bundles are transferred in response to explicit client requests. Ideally, each bundle will consist entirely of files that are always loaded together. In addition, the files in the bundle should be ordered such that they match the order of requests by the client.

As in Jar file downloading, the bundles are precompressed, so no compression needs to be performed on the server at runtime. This allows the server to scale more easily to a large number of clients.

As in on-the-fly compression (and unlike Jar files), compression is performed on multiple files instead on individual files. This allows more opportunities to share redundant information between files, resulting in a better compression ratio than with individual compression.

2.1 Dividing a collection of files into bundles

The main problem in applying bundling to network class loading is determining how to divide the collection of files needed by applications into bundles. In order to ensure that the client generally receives only files it needs, we must only put two files in the same bundle if those files are always (or almost always) needed together in the same program execution. Furthermore, the files in the bundle should always (or almost always) be ordered in the same order that they will be requested by the client. When these two properties hold, bundling has all of the desirable properties of on-demand class loading, with the additional benefits of compression and pre-sending.

Clearly, to achieve a good division of files into bundles, we need to have knowledge of a typical program's class and resource loading behavior at runtime. We chose to use *class loading profiles* as the source of information about program behavior. Class loading profiles record the order and time at which each class or resource was loaded during execution.

2.2 Conceptual framework

We chose to view the collection of files as a fully connected weighted graph. Each file is represented by a node in the graph. The weight of an edge from file A to file B represents the desirability of placing A and B in the same bundle.

Given a set of profiles, there are many ways to determine the edge weights. We chose to use *frequency correlation* as the basis for the edge weights. The frequency correlation of two files A and B is defined as t/n, where t is the number of profiles in which both A and B are loaded, and n is the number of profiles in which either A or B is loaded. A frequency correlation of 1.0 indicates that A and B are always loaded together.

The bundling algorithm considers edges according to an order produced by the *edge comparator*. We chose to use weight as the primary criterion for the edge comparator, and average distance as the secondary criterion. (The average distance of an edge connecting files A and B is the average distance between A and B in the input profiles.) This sort order helps ensure that edges connecting files usually loaded close to each other are considered before edges connecting files usually loaded far apart.

While frequency correlation is a good measure of how often two files are loaded together, it is not a measure of whether those files are generally loaded near each other in the profiles. If we put two files which are generally loaded far apart from each other in the same bundle, if a request is made for one of the files the other will be loaded too early. To ensure that all of the files in a bundle are loaded near each other, we limit the maximum *bundle spread*. The bundle spread for a proposed bundle b is the maximum over all profiles p of

$$\text{lastMoment}(b, p) - \text{firstMoment}(b, p) - \text{size}(b) + 1$$

The ‘moment’ of a file in a profile is the ordinal value indicating when it was loaded relative to the other files in the profile. (I.e., the first file in the profile is moment 0, the second is moment 1, etc.) The best possible spread for a bundle is 0, indicating that for any profile in the input set, after any file in the bundle is requested all of the files in the bundle will be used before any file not in the bundle.

In addition to determining which files to bundle together, the bundling algorithm must also determine the order of the files within each bundle. This order is determined by a *bundle sort comparator*. We chose to use *average position* as the criterion for the bundle sort comparator. Given a bundle b and a profile p, the position of each file is found in relation to the other files in b. For example, if A is loaded in p before any other files in b, then its position is 0. The average position of file A in a bundle b is simply its average position over all input profiles. Using average position as the criterion for the bundle sort comparator helps ensure that files are placed in the order in which they will be needed, based on the order in which they occurred in the input profiles.

2.3 Bundling algorithm

Once the input profiles have been used to create the graph, the bundling algorithm uses the graph to put the files into bundles. The algorithm works as follows:

1. Initially, each file is put in a separate bundle.
2. Edges whose weight is less than the *minimum edge weight* are discarded.
3. The edges are sorted according to the *edge comparator*.
4. The edges are processed one at a time, in the order determined in step 3. For each edge, if the files connected by the edge are not already in the same bundle, and if the number of files in the resulting bundle would not exceed the *maximum bundle size*, and if the resulting bundle would not exceed the *maximum bundle spread*, then the bundles containing the files are combined into a single bundle.
5. After all the edges have been processed, the files within each resulting bundle are sorted according to the *bundle sort comparator*.

Once the contents of the bundles have been determined, they are combined and compressed cumulatively. We evaluated both zlib (at the highest compression level) and Pack as compressed formats for the bundles.

2.4 Implementation

The bundle class loader consists of a *server* and a *client*.

The server accepts requests for files from the client. For each request, it sends the bundle containing the requested file. Because the bundles are pre-compressed, the server does not perform any compression at runtime. As a result, the server has little CPU overhead.

The client consists of an implementation of `java.lang.ClassLoader`. When a call to `loadClass()` or `loadResource()` is made to the class loader, it first checks to see if the data for the requested class or resource has already been received. If so, it handles the request using the data already received. Otherwise, it sends a request for the needed file to the server. In return, it receives a bundle from the server containing the requested file. Because the bundle generally contains files not explicitly requested by the client, the client caches the data for all files received in memory. Once the data for a class or resource has been used by the client, its cache entry is deleted, so that the memory can be reclaimed by the garbage collector.

The implementation of the client and server are written in Java, consisting of 733 lines of source overall. The total size of the client class files is approximately 20 KB. Both client and server use only the standard Java runtime classes in their implementation, and will work with any JVM compatible with Sun’s JDK 1.2.

Our implementation currently supports only bundles compressed with zlib. Future work will add support for bundles in Pack format.

2.5 Latency issues

Network latency has a substantial effect on the performance of any request-driven file transfer protocol, since it causes a delay between the time the client issues a request for a file and the time when the data for that file starts to arrive. This is one of the principal disadvantages of purely on-demand transfer strategies, such as downloading individual files. The request latency problem is compounded when the available bandwidth is high, since the penalty associated with each request is the product of the latency and the bandwidth (since the product is the amount of data that can be transferred during one latency period).

To decrease the penalty associated with network latency, it is necessary to reduce the number of requests issued by the client. The simplest way to do this in bundling is to find ways to increase the bundle size. In our experiments, we tested a range of bundling parameters, with the goal of producing bundlings with a large average bundle size. Section 3 will show how well we were able to counteract the effects of network latency.

3 Experimental results

This section describes experiments we performed to evaluate the effectiveness of bundling compared to other network class loading mechanisms.

There are two fundamental ways to apply bundling. First, it may be applied to the class and resource files used by a single application, based on profiles collected from that application. This approach is reasonable if the intent is to optimize the transfer of files used by a single application in isolation. The second way it can be applied is to a library used by many applications. Applying bundling to a library used by multiple applications is a more difficult problem than applying it to a single application, because the class loading behavior of different applications is much less consistent than the class loading behavior of multiple runs of a single application. (Note that there is no absolutely rigid distinction between these approaches. For example, an office application might have a common library of classes as well as several ‘sub-applications’, such as word processor, spreadsheet, etc. The loading behavior for the classes and resources within the sub-applications would likely be quite consistent from run to run. However, each sub-application might use the common library in different ways, resulting in inconsistent loading behavior within the classes and resources of the library from run to run.)

In order to test both extremes (multiple applications *vs.* single application), we generated bundlings from a subset of JDK 1.2.2’s `rt.jar`, which contains the standard Java libraries used by all applications. The subset includes AWT, Swing, and Java2D, but does not include the ‘core’ packages (`java.lang.*`, `java.util.*`, etc.) Since almost all interactive Java applications use AWT, Swing, or Java2D, we felt the `rt.jar` subset was a good example of a substantial library used by many applications in nontrivial ways.

The first experiment uses profiles from several applications and applets to generate bundlings for the subset of `rt.jar`. Since the applications load different class files and resources in significantly different orders, this experiment represents a ‘stress test’ of the bundling algorithm. We would like to emphasize that applying bundling to a large, multi-purpose library is not an ideal application of bundling, since bundling is designed to take advantage of regularities in class loading behavior.

The second experiment uses profiles from a single application to generate bundlings for the `rt.jar` subset. Since there is more regularity in the class and resource loading behavior of a single application than in several different applications, this experiment represents an ideal scenario for bundling, in which we would expect its performance to come close to that of cumulative compression in terms of the compression ratio achieved, while avoiding the high latency delays associated with on-demand loading.

The third experiment measures the effectiveness of the bundlings generated in the first experiment on two applications not represented in the input profiles used to generate those bundlings. This experiment is an even less ideal application for bundling than the first experiment; however, it does offer some insight into how the performance of bundling may degrade when it encounters class loading behavior not represented in the input profiles.

The fourth experiment measures the startup time for one of the test applications using simulated network bandwidth and latency. The experiment is designed to determine how well bundling performs in a real JVM, as opposed to off-line simulations.

3.1 Bundling parameters

In generating the bundlings for the experiments, we tried three combinations of bundling parameters:

- Minimum edge weight 1.0, maximum bundle size 200, maximum bundle spread 5. These bundling parameters are relatively strict, in the sense that they forbid two files from being placed in the same bundle if there were any profiles in which one was loaded without the other. They also significantly limit the extent to which files can be delivered ahead of when they are needed. These parameters produced a bundling where the average bundle size was quite small, on the order of 3 or 4 files.
- Minimum edge weight 0.8, maximum bundle size 1000, maximum bundle spread 200. This is a ‘looser’ set of parameters, which permits files to be placed in the same bundle even if they aren’t always loaded together, and also allows files to be sent significantly ahead of when they are needed. These parameters produced a bundling where the bundles were larger, the average bundle size being about 12 files.
- Minimum edge weight 0.8, maximum bundle size 1000, maximum bundle spread 500. This is an even looser set of parameters, which produced bundles whose average size was about 24 files.

Note that in all cases, the maximum bundle size parameter was not reached, so bundle growth was only constrained by the minimum edge weight and maximum bundle spread.

In the figures and text, the parameters used to generate a bundling are specified in the form w-m-s, where w is the minimum edge weight, m is the maximum bundle size, and s is the maximum bundle spread.

3.2 Network parameters

We tried two combinations of network bandwidth and latency in our simulations. First, we tried 500,000 bytes/second bandwidth and 4 ms latency, which is typical of what one might see on a wireless local area network. Second, we tried 50,000 bytes/second bandwidth and 70 ms latency, which is typical of a high-speed wide area network.

3.3 Measurements

Three types of data were collected.

First, for experiments 1–3, we performed a simulation of expected arrival times for each file loaded in the test profiles, based on network bandwidth and latency. The idea is to assume that the client issues requests for each file in the profile one at a time, and that the requested file must arrive before the next file in the profile can be requested. A request for a file which has not arrived yet must be sent to the server. Only one bundle may be in transit at any given moment, and the data for a requested bundle starts arriving only after the latency period has elapsed following the request, or after all bundles in transit have finished transferring, whichever is later. We take into account the relative offsets of each file within a bundle, and their sizes within the bundle. While this model is somewhat simplistic, it does take into account the most important factors in network class loading, since the computational overhead of network class loading is generally less significant than the network overhead. The expected arrival time for each file in the test profile is compared with the 'ideal' arrival time, which is simply the time the file would arrive if a single bundle containing all of the files in the profile in the correct order were downloaded. (This is like on-the-fly compression, except that the entire sequence of requests is 'known' in advance.)

Second, for the applications in experiment 1, we measured the total number of bytes transferred from server to client. The fewer bytes transferred, the higher the compression ratio and the less bandwidth consumed. Note that even though our current implementation supports only bundles in zlib format, we were able to accurately measure the download sizes for Pack bundles based on the bundle transfer behavior observed for zlib bundles.

Third, in experiment 4 we measured the startup time for one of the test applications (Argo/UML) under simulated network bandwidth and latency conditions. By running an actual JVM under realistic conditions, we are able to see how successfully our estimates of class loading performance (the expected arrival times described above) predict real application performance. In particular, this experiment takes into account other sources of overhead in the JVM, such as loading native libraries, verifying class files, etc.

3.4 First experiment

In the first experiment, bundlings of the `rt.jar` subset were generated using a collection of 17 input profiles, from the following applets and applications:

- Sun's Java2D demo
- Argo/UML, an object-oriented design tool
- Sun's Swing demo
- The JDK 1.2.2 demo applets
- The Jazz HiNote zoomable user interface demo (from the University of Maryland's Human-Computer Interaction Lab)
- The jEdit editor, version 2.3pre2

We evaluated the bundlings produced from the input profiles on five applications: the Drawtest applet, the TicTacToe applet, Argo/UML, the Java2D demos, and HiNote. For all five applications, the profile used in the experiment was a member of the input profiles used to generate the bundlings.

Figures 1–5 show the expected file arrival times vs. ideal file arrival times for the applications. In general, the bundlings were competitive with the ideal arrival times, although the 'strict' bundling 1.0-200-5 fared noticably worse than the 'loose' bundlings. This shows that even when network latency is relatively low (4 ms) it is still an important factor.

The vertical 'cliffs' in the plots correspond to files being sent earlier than when they are needed; essentially, they delay the file whose request is currently outstanding. The diagonal sections (where the slope diverges from the ideal) corresponds to request latency. The horizontal 'plateaus' in the plots correspond to files already received being used by the client. These satisfy their requests instantaneously, since the data is already available.

Figure 6 shows the download sizes for the applications and bundlings in experiment 1, for bundles in both zlib and Pack formats. All of the zlib bundlings were competitive with cumulative zip, although they were much larger than an equivalent Pack archive would have been. The 'loose' Pack bundlings (0.8-1000-200 and 0.8-1000-500) were able to get quite close to the compression ratio of a single Pack archive, while the 'strict' Pack bundling (1.0-200-5) had somewhat lower compression than a single Pack archive. All of the Pack bundlings were significantly better than cumulative zip.

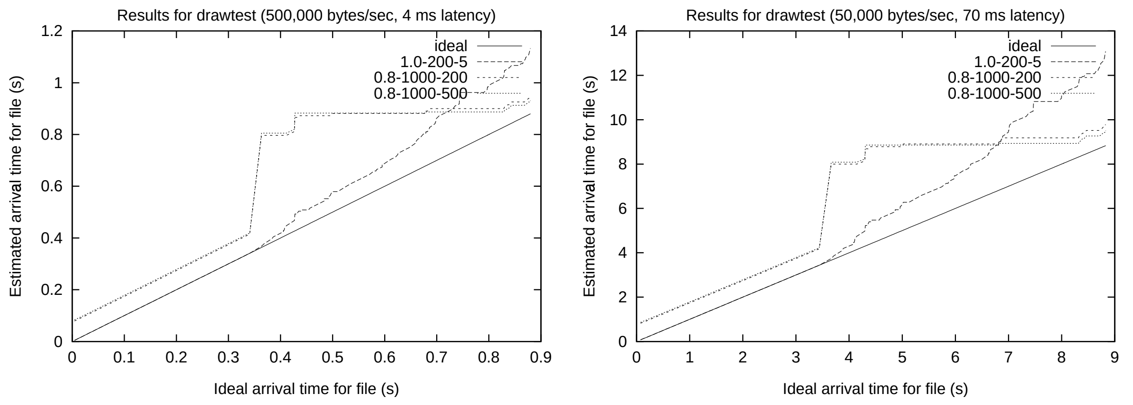

Figure 1: Ideal file arrival times vs. estimated file arrival times for drawtest (experiment 1).

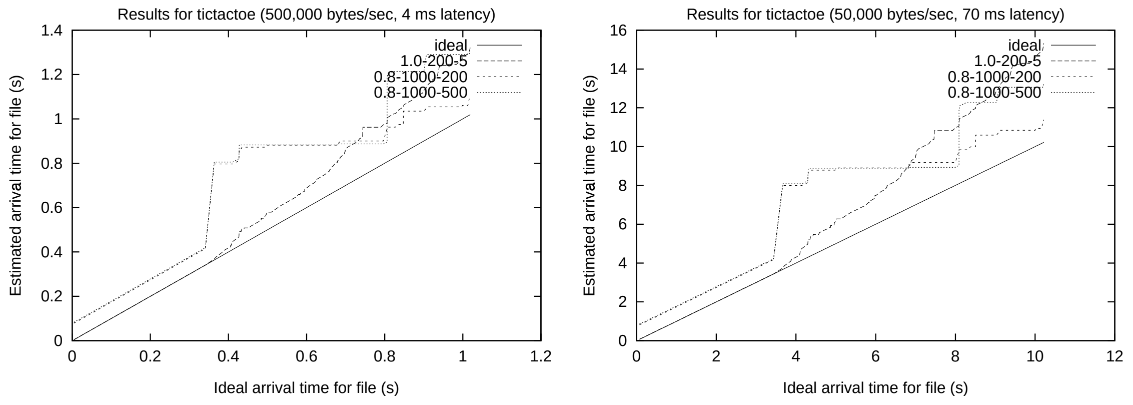

Figure 2: Ideal file arrival times vs. estimated file arrival times for tictactoe (experiment 1).

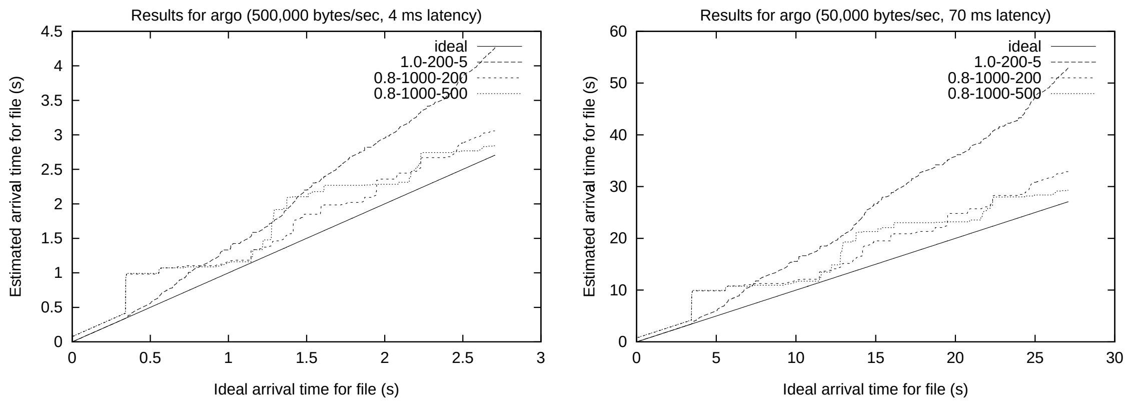

Figure 3: Ideal file arrival times vs. estimated file arrival times for Argo/UML (experiment 1).

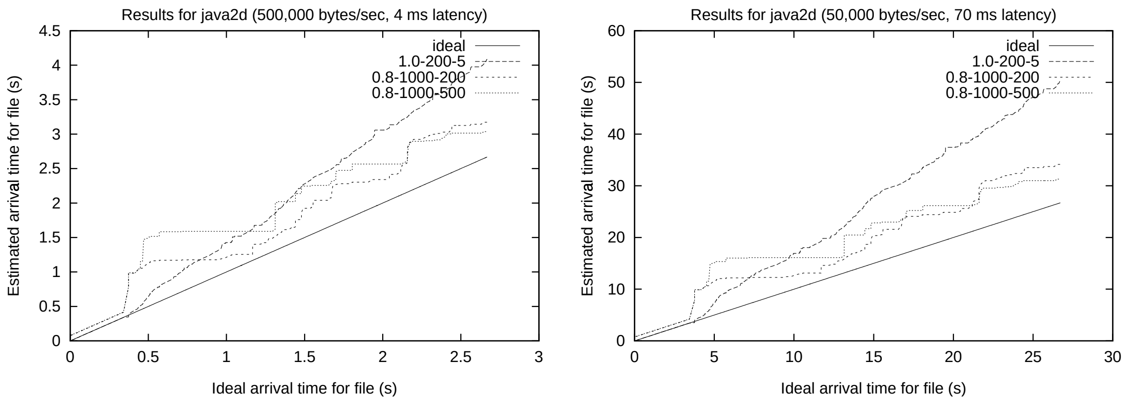

Figure 4: Ideal file arrival times vs. estimated file arrival times for Java2D (experiment 1).

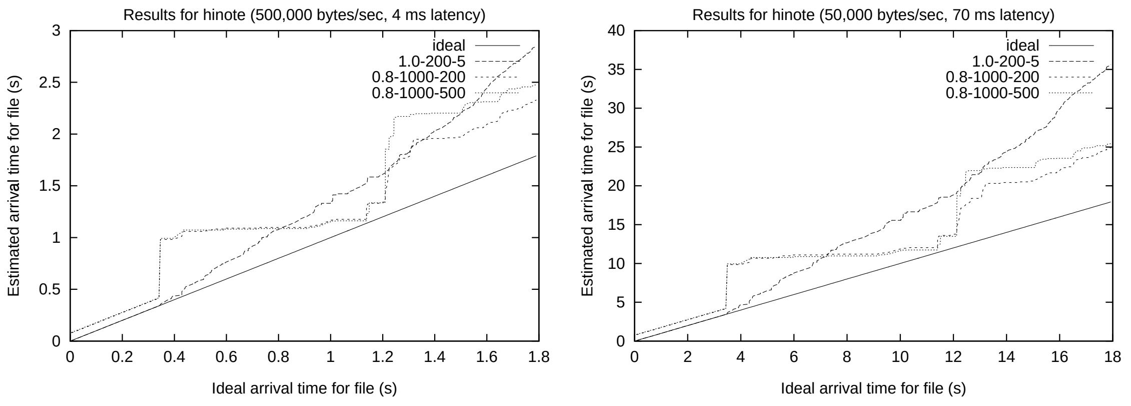

Figure 5: Ideal file arrival times vs. estimated file arrival times for HiNote (experiment 1).

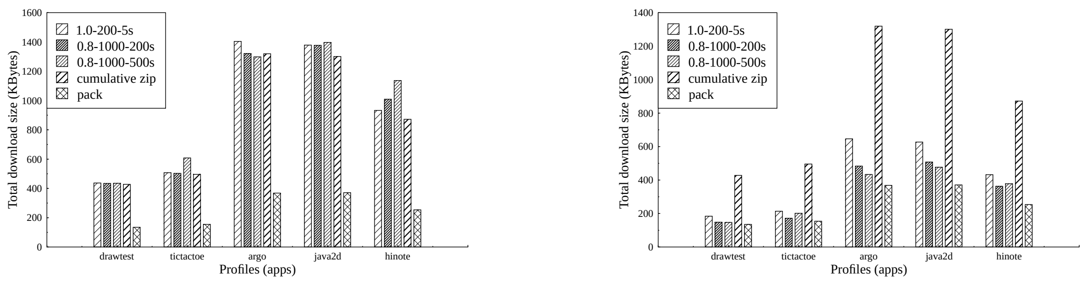

Figure 6: Download sizes for zlib bundles and Pack bundles vs. cumulative zip and Pack (experiment 1).

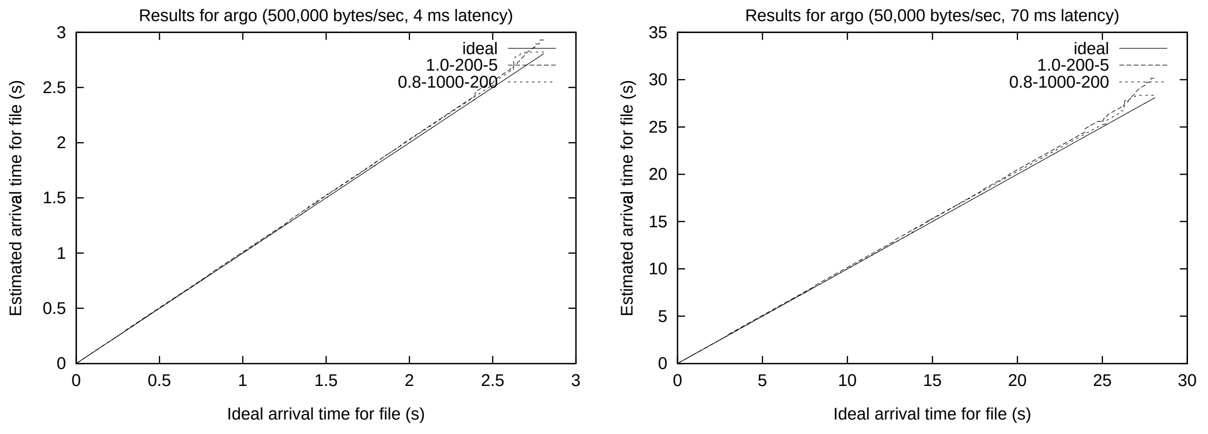

Figure 7: Ideal file arrival times vs. estimated file arrival times for Argo/UML (experiment 2).

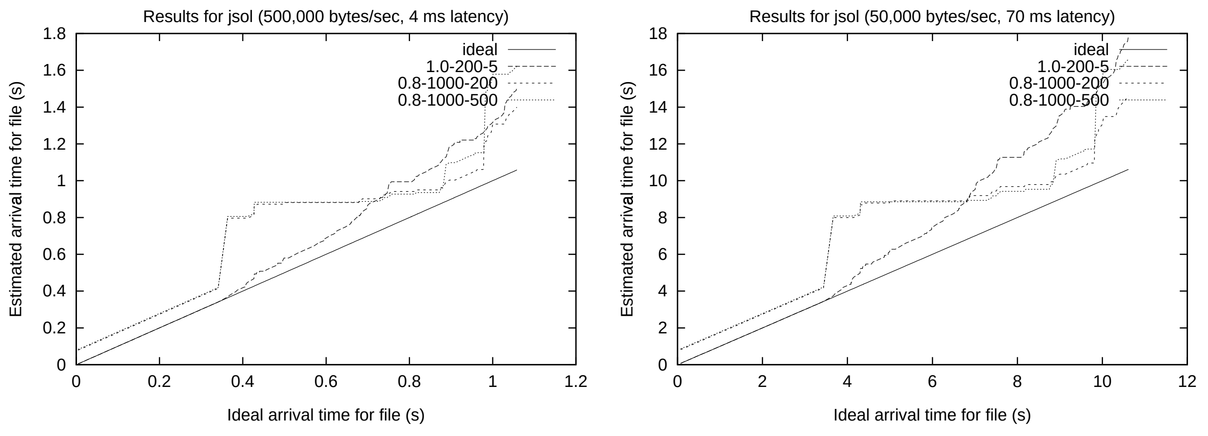

Figure 8: Ideal file arrival times vs. estimated file arrival times for jsolitaire (experiment 3).

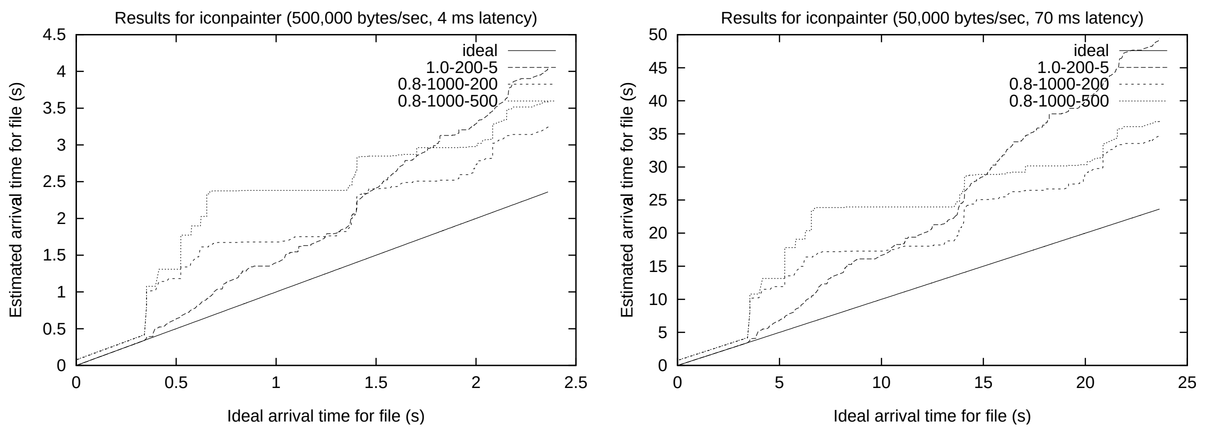

Figure 9: Ideal file arrival times vs. estimated file arrival times for iconpainter (experiment 3).

Minimum edge weight	Maximum bundle spread	Number of bundles	Startup time (seconds) 500,000 bytes/s 4 ms latency	50,000 bytes/s 70 ms latency	Number of unused files transferred
n/a	n/a	1	22.47	44.74	0
1.0	5	317	24.54	67.77	0
0.8	200	88	22.57	48.85	57
0.8	500	30	22.50	46.46	99

Table 2: Startup time for Argo/UML under simulated network bandwidth and latency.

3.5 Second experiment

In the second experiment, bundlings of the `rt.jar` subset were generated from 5 profiles collected from a single application, Argo/UML. Each profile exercised different application functionality. Then, arrival time data were collected for a single profile (which was a member of the profiles used to generate the bundlings).

Figure 7 shows the expected file arrival times vs. ideal file arrival times. As expected, the bundlings achieve file arrival times very close to the ideal file arrival times. Because of the regularity in class loading behavior in different runs of the same application, only a few bundles were generated, increasing the bundle size and compression ratio, and decreasing the cost associated with making requests to the server.

(Note that bundling 0.8-1000-500 is not shown because it is identical to 0.8-1000-200 for the profiles used.)

3.6 Third experiment

In the third experiment, we measured the effectiveness of the bundlings of the `rt.jar` subset produced in experiment 1 on two applications not represented in the input profiles used to generate those bundlings. The applications were jsolitaire (a Solitaire card game applet) and iconpainter (an icon editor). Note that the iconpainter used four class files not loaded in any of the original input profiles; we added these to the bundlings as bundles of size 1 (i.e., containing a single class file).

Figures 8 and 9 show the expected vs. ideal file arrival times for jsolitaire and iconpainter, respectively. The arrival times for the bundlings are somewhat further from ideal than in experiment 1, where the profiles tested were members of the set of profiles used to generate the bundlings. This shows that bundling works best for applications which are represented in the input profiles used to generate the bundlings. However, the time for all of the files to arrive for the loose bundlings is only about 40% later than for the ideal case. Considering that bundling typically achieves twice the compression of Jar archives, this overhead may be acceptable in some situations.

3.7 Fourth experiment

In the fourth experiment, we measured the amount of time needed by the Argo/UML application to initialize and display its user interface, using the bundle class loader client and server for the `rt.jar` subset. We simulated network bandwidth by restricting the rate at which the server sent data, and we simulated network latency by instrumenting the server to pause for a fixed amount of time before processing a client request. The client and server communicated over local TCP/IP on a two-processor Sun Ultra 60 workstation.

Table 2 shows the results. The first row of the table gives the startup times for a single ‘ideal’ bundle consisting of all of the required files in the correct order. The other rows show the startup times for the bundlings used in experiment 1. For the high bandwidth and low latency case, the startup times for the bundlings were almost identical to the ideal startup time. For the lower bandwidth, higher latency case, the ‘loose’ bundlings performed much better than the ‘strict’ bundling, even though some unneeded files were transferred.

4 Related work

Pack [9] is a compressed archive format for Java class files developed by William Pugh. While it achieves very good compression, it has a relatively slow decompressor, so is most useful for slow networks.

Jazz [3], developed by Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek, is another compressed archive format for Java class files. It is similar to Pack, but uses a somewhat less aggressive compression scheme.

In [11], Tip, Laffra, Sweeney, and Streeter describe Jax, an application extractor for Java. Jax uses a number of whole-program analysis techniques to transform an application’s class files to reduce its size. For example, Jax detects and eliminates unused methods and fields.

One motivation for size reduction is reducing download time when loading the application from a network. The techniques employed by Jax are largely orthogonal to compression and bundling, and would be complementary to the techniques we present in this paper.

In [6], Krintz, Calder, and Hölzle describe techniques for reducing transfer delay by splitting classes (methods and data members) into hot and cold parts, and for prefetching classes. The class file splitting technique transforms classes to avoid downloading code for rarely used methods and fields. The prefetching technique tries to avoid request latency and transfer delay by placing requests for class files prior to the 'first-use' points for those classes. Because these techniques rely on the client issuing an explicit request for each file, they would only by suitable for use with individual file compression or on-the-fly compression.

Appwerx Expresso [2] is a commercial product that reduces applet startup time by reordering class file and resources in a Jar file and implementing a class loader that can concurrently download and execute classes from the Jar file.

In [7], Kuenning and Popek describe a technique for hoarding files on a mobile computer prior to disconnection from a network. Although the problem they address is different, their approach is similar to ours in that they use profiles of file access by the user to group related files together and to predict which files will be needed by the user. However, there are substantial differences between their techniques and ours. Their notion of 'semantic distance' between two files is based on the number of intervening file accesses to other files in the profiles, and on the relative orders of open and close operations on files, while our notion of edge weight is based on the frequency of the files being loaded in the same profile. (Because their profiles are system-wide, they do not have the notion of profiles specific to an application.) Whereas our motivation for grouping files into bundles is to enable better compression and for pre-sending to reduce request latency, their motivation is to detect related groups of files in order to ensure that entire 'projects' are hoarded, rather than just recently used files in isolation.

5 Conclusions

We described a technique called *bundling* which splits a collection of class files and resources into bundles based on previously observed class and resource loading behavior. We showed that bundling is competitive with cumulative compression when the applications and profiles are known in advance, and that it is no worse than the Jar archive format when used on an application not included in the training set.

One possibility for future work is to use a static dictionary of commonly occurring class file data in compressing the bundles. This could improve the compression ratio on small bundles.

Another possibility for future work is to apply bundling in other contexts where the transfer of sequences of files is required. For example, it might be applicable in serving a collection of files to be partially mirrored at other sites.

Acknowledgements

We would like to thank Jeremy Manson for his helpful comments and suggestions. We would also like to thank the anonymous reviewers for identifying weaknesses in the original version of the paper.

Notes

[1]Java is a trademark of Sun Microsystems.

[2]Note that for on-the-fly compression to work, the compression algorithm must have the ability to flush the compressed stream at arbitrary points (i.e., the file boundaries). Zlib has this ability, but it is not implemented in the `java.util.zip` package.

[3]For HTTP 1.1 persistent connections.

[4]Not implemented in Java HTTP client.

[5]Jar indexes can support an arbitrary request granularity.

References

[1] zlib Home Site. http://www.info-zip.org/pub/infozip/zlib/.

[2] Appwerx, Inc. Appwerx Expresso. http://www.appwerx.com/expresso/.

[3] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An Efficient Compressed Format for Java Archive Files. In *Proceedings of CASCON'98, Toronto, Ontario*, 1998.

[4] L. P. Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3, May 1996.

[5] L. P. Deutsch and J-L. Gailly. RFC 1950: ZLIB compressed data format specification version 3.3, May 1996.

[6] Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Denver, Colorado*, 1999.

[7] Geoffrey H. Kuenning and Gerald J. Popek. Automated Hoarding for Mobile Computers. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[8] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, Canada*, 1998.

[9] William Pugh. Compressing Java Class Files. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, Georgia*, 1999.

[10] Sun Microsystems. JDK 1.3 — JAR File Specification, 1999.

[11] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical Experience with an Application Extractor for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Denver, Colorado*, 1999.