More Efficient Network Class Loading through Bundling

David Hovemeyer and William Pugh
Department of Computer Science
University of Maryland

Outline

- Motivation
- Algorithm
- Implementation
- Experimental results
- Conclusions

Motivation

- Network class loading is important
  - ▷ The web
  - ▷ Wireless computing
  - ▷ Thin clients
- Want to minimize application startup time and runtime delays
- Existing mechanisms (Jar archives, on-demand) have some shortcomings

Goals

What properties would we like ideally?

- Transfer as few bytes as possible, to make best use of available bandwidth
- Files arrive when needed, in the correct order
- Limit number of requests by client (to reduce request latency costs)
- System should be scalable and easily deployed

Archive formats

- Examples: Jar, Pack
- Advantages:
  - ▷ Only one request must be sent in order to get the entire application
    - so request latency cost paid only once
  - ▷ Contains a large number of files, so more opportunities for compression
- Disadvantages:
  - ▷ The archive may contain files which won't be needed
  - ▷ The files may be in the wrong order

Jar file limitations

- Jar archives have some specific limitations when used for network class loading
  - ▷ Files are compressed individually, so opportunities for reuse (better compression) are missed
  - ▷ `URLClassLoader` waits for entire archive to be transferred before loading any class
- These limitations related to use of Jar files as an on-disk format
- For example, individual compression allows random access to files

On-demand class loading

- *E.g.*, loading individual files relative to a directory URL
- Advantages:
  - ▷ Only files that are needed are transferred
  - ▷ Files arrive in correct order
  - ▷ In principle, could use cumulative compression
- Disadvantages:
  - ▷ Must pay request latency cost for every file!
    - ◦ Could be 100's of milliseconds per request
  - ▷ Compressing on the fly takes a lot of CPU time — not scalable

Prefetching

- Prefetching can be used to hide request latency in on-demand loading
  - ▷ Calder, Krintz, and Hölzle, *Reducing transfer delay using Java class file splitting and prefetching*, OOPSLA 1999.
- Files may be requested in any order, so cumulative compression would be difficult

A hybrid approach

- Can we combine the desirable properties of archives and on-demand loading?
  - ▷ Try to avoid downloading files that aren't needed
  - ▷ Try to get files in correct order
  - ▷ Use files as soon as they arrive!
- Transfer granularity should be large enough to
  - ▷ Reduce the effects of request latency
  - ▷ Increase compression ratio
- *Idea*: create 'bundles' of files

Bundling

- Divide the collection of files into bundles:
  - ▷ Avoid putting files that aren't needed together in the same bundle
  - ▷ But otherwise make them as large as possible
- Use class and resource loading profiles to determine how to divide the files
  - ▷ ... assuming that past behavior is a good predictor of future behavior

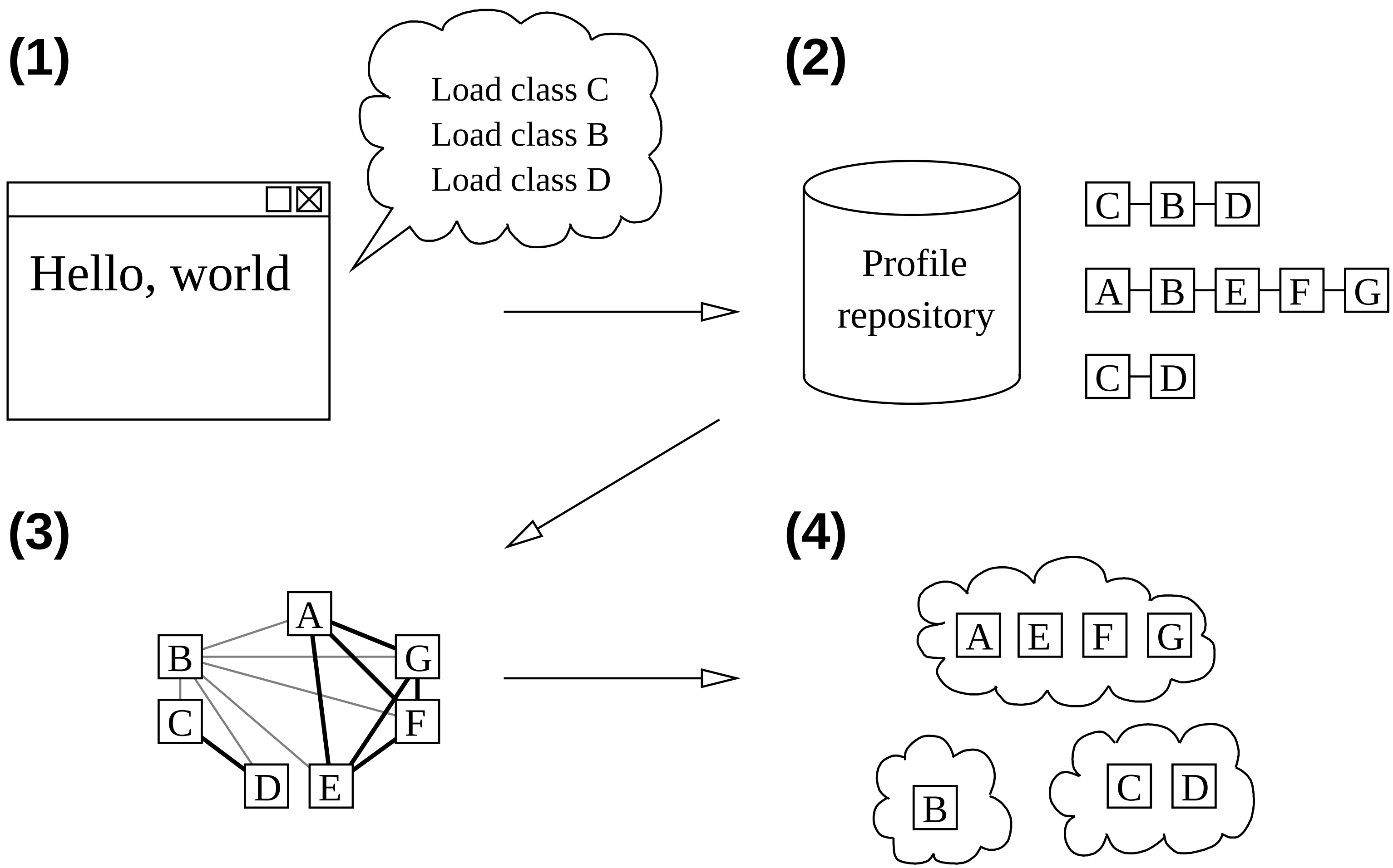
Outline

- Motivation
- Algorithm
- Implementation
- Experimental results
- Conclusions

Algorithm — Goals

- A bundle is a compressed sequence of files
- Compression is cumulative
- Goal of algorithm is to create bundles such that
  - ▷ If the bundle is downloaded, all (or most) of its files will be needed
  - ▷ Its files are (mostly) in the correct order
- The bundles should be as large as possible, as long as they satisfy the above criteria

Algorithm — Overview



Graph

- The collection of files (classes and resources) is represented by a weighted graph
  - ▷ Nodes represent the files
  - ▷ Edges weights represent the likelihood that the files connected will be needed in the same program execution

- Edge weights are determined by *frequency correlation*
  - ▷ For files *A* and *B*, defined as n/t
  - ▷ n is the number of profiles in which both *A* and *B* are loaded
  - ▷ t is the number of profiles in which either *A* or *B* are loaded

- Edge weight of 1.0 means files always loaded together (in profiles)

Edge sort comparator

- The algorithm considers the edges of the graph one at a time, to determine if the files connected should be placed in the same bundle

- Two-level sort:

  1. First by *weight*
  2. Next by *average distance* between the files connected by the edge

- Consider strongly correlated files before more weakly correlated files

- Consider files generally close together before files that are farther apart

- Other sorting criteria are possible

Bundle spread

- Ideally, the files in a bundle are needed at the same time
- Use *bundle spread* metric to prevent bundles from containing files loaded far apart
- For bundle b and profile p,

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

- Bundle spread of bundle b is maximum $\text{spread}(b,p)$ over all input profiles p
- 'Ideal' bundle spread is 0, meaning all files in bundle will be used before any files not in the bundle (according to profiles)

Bundle sort comparator

- Once the algorithm has decided which files to bundle together, need to order them
- Want to deliver them close to the order expected by the application
- Sort files by their *average position* in the profiles
  - ▷ Normalized for each profile by position of earliest file in the bundle

Algorithm

- Each file starts out in a separate bundle
- Discard edges where weight $<$ *minimum edge weight*
- Sort edges according to *edge sort comparator*
- For each edge connecting files A and B, if

  1. A and B not already in same bundle, and
  2. resulting bundle would not exceed *maximum bundle size*, and
  3. resulting bundle would not exceed *maximum bundle spread*

  then the bundles containing A and B are combined.

- Bundles are sorted according to *bundle sort comparator*

Outline

- Motivation
- Algorithm
- Implementation
- Experimental results
- Conclusions

Implementation

- Analysis of profiles and creation of bundles done off-line
- Bundles compressed with zlib (`java.util.zip.*`)
  - ▷ We used zlib because it is part of the standard Java libraries, is stream-oriented, and has a fast decompressor
  - ▷ However, other compressed formats could be used
  - ▷ *E.g.*, the Pack format (Pugh, *Compressing Java Class Files*, PLDI 1999)
- Specialized client and server written in Java
  - ▷ Less than 1000 lines of code total
  - ▷ Implemented using standard Java 1.2 API
  - ▷ Client uses customized class loader

Outline

- Motivation
- Algorithm
- Implementation
- Experimental results
- Conclusions

Experiments

- Four experiments
  - ▷ Experiments 1 and 4: Stress test — profiles from many applications
  - ▷ Experiment 2: Realistic case — profiles from one application
  - ▷ Experiment 3: Test of application not represented in input profiles
- All experiments test the loading of a subset of JDK 1.2.2 `rt.jar`
  - ▷ Contains AWT, Swing, Java2D
  - ▷ Not 'core' classes (`java.lang.*`, etc.)
- Note that bundling is applied to the *library*, not the application

Measurements

- Simulated file arrival time, taking into account bandwidth and latency (experiments 1, 2, and 3)
  - ▷ For each request, schedules a bundle transfer and calculates file arrival times
  - ▷ Compared with arrival times for single 'ideal' bundle consisting of all requested files, in order
  - ▷ For two bandwidth/latency combinations
- Total number of bytes downloaded (experiment 1)
- Application startup time in a real JVM (experiment 4)

Bundling parameters

Minimum edge weight	Maximum bundle size	Maximum bundle spread	Abbrev.
1.0	200	5	1.0-200-5
0.8	1000	200	0.8-1000-200
0.8	1000	500	0.8-1000-500

- 1.0-200-5 is a ‘strict’ bundling — few unneeded files or mis-orderings, smaller bundles

- 0.8-1000-200 and 0.8-1000-500 are ‘loose’ bundlings — more unneeded files sent, larger bundles

Why create multiple bundles?

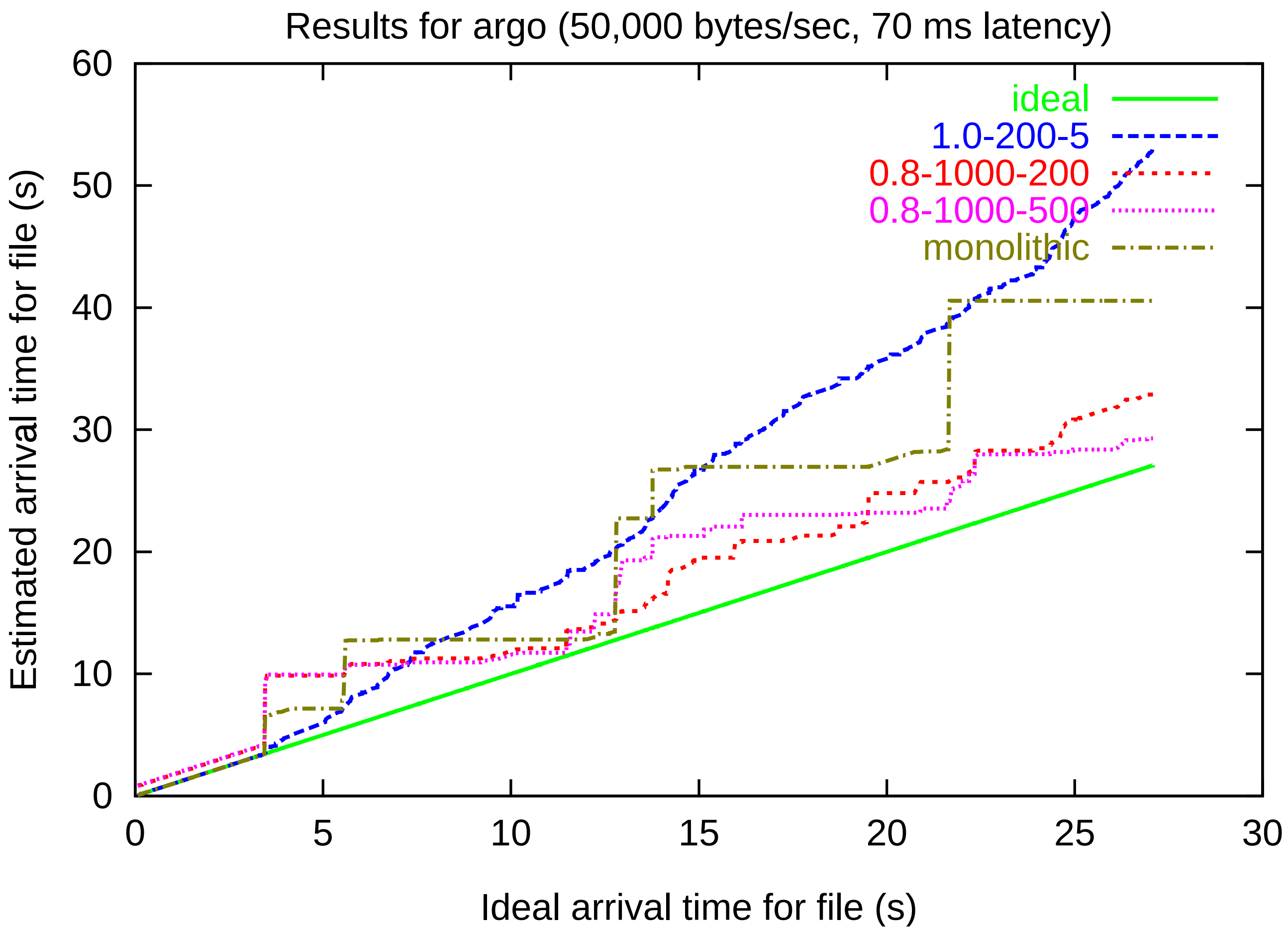
- Couldn't we just put all files in a single bundle, like a Jar file?
- Get advantages of cumulative compression
- To this end, created a 'monolithic' bundling, consisting of all files in a single bundle, sorted by average position (not in paper)
- Not in paper

Experiment 1

- A 'stress test'
- 17 input profiles collected from 5 applications and several applets on the `rt.jar` subset
- The applications had considerably different loading behaviors
- Note: this is not the way bundling is intended to be used in a 'real' application
- Tests done on profiles which were members of the input set

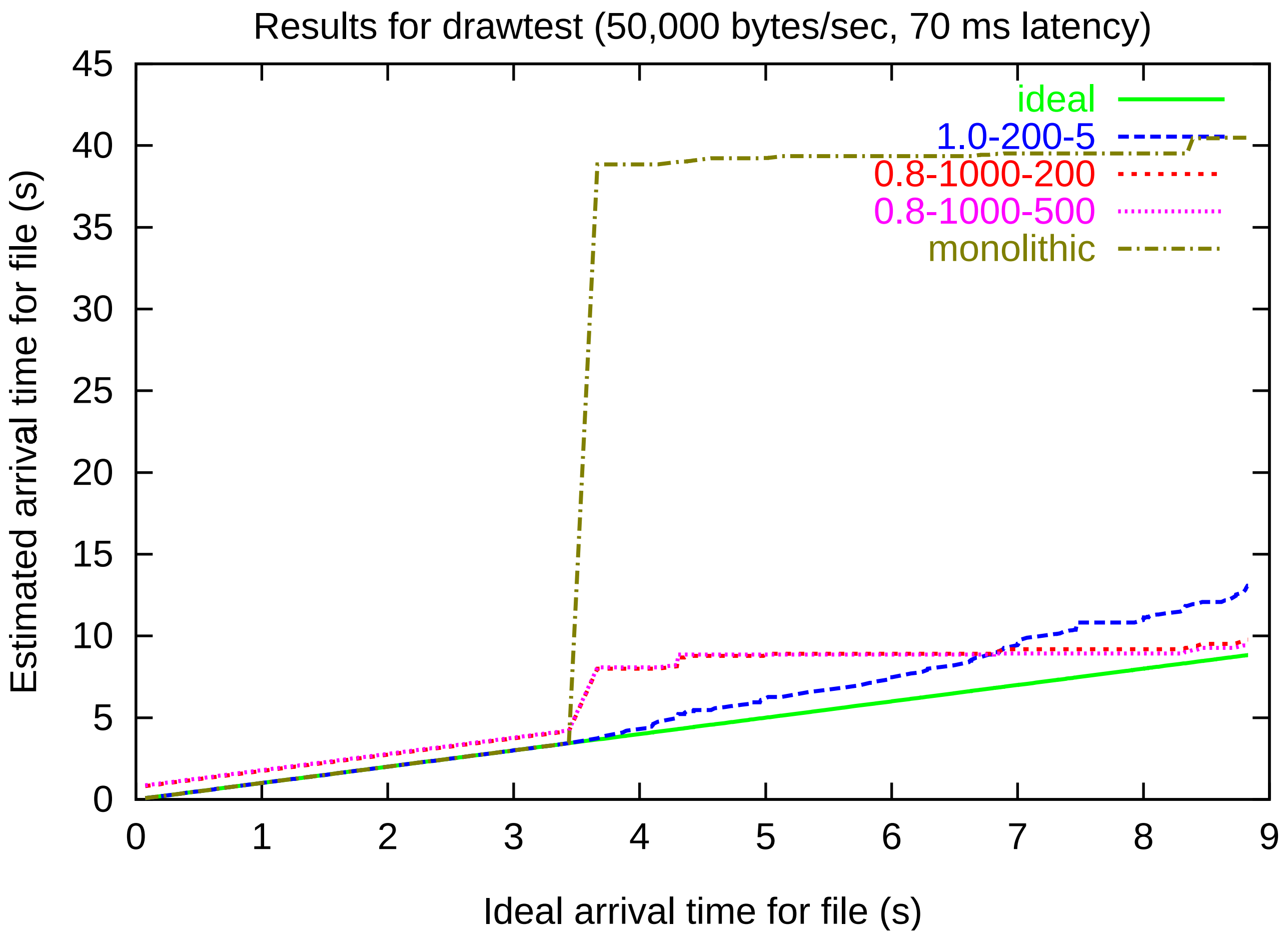
Experiment 1

Expected file arrival times vs. ideal for Argo/UML: 50,000 bytes/second bandwidth, 70 milliseconds latency



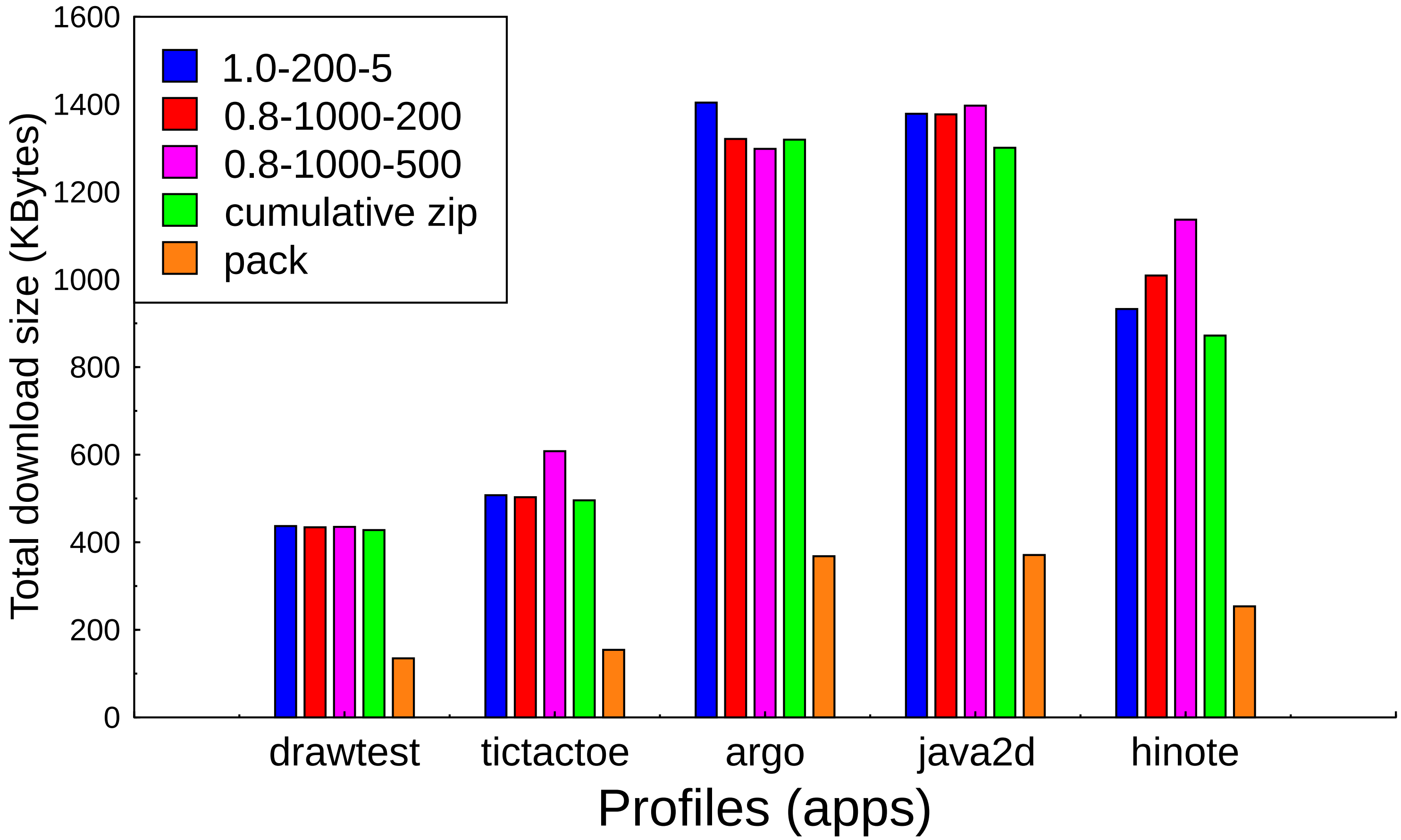
Experiment 1

Expected file arrival times vs. ideal for drawtest: 50,000 bytes/second bandwidth, 70 milliseconds latency



Experiment 1

Number of bytes downloaded for Argo/UML (zlib bundles)



Experiment 4

- Measure application startup time for Argo/UML using bundlings from experiment 1
- See how bundling performs in a real JVM
- Setup:
  - ▷ Restrict transfer rate to simulate network bandwidth
  - ▷ Add delay to server to simulate network latency
  - ▷ Run on 2-processor Sun Ultra 60 over local TCP/IP
  - ▷ JDK 1.2.2, HotSpot
- Compare with startup time for 'ideal' Jar file and `URLClassLoader` (not in paper)

Experiment 4

Delivery	Number of bundles	Startup time (s)	Number unused files transferred
‘ideal’ bundling	1	44.74	0
‘ideal’ jar file	1	51.63	0
1.0-200-5	317	67.77	0
0.8-1000-200	88	48.85	57
0.8-1000-500	30	46.46	99

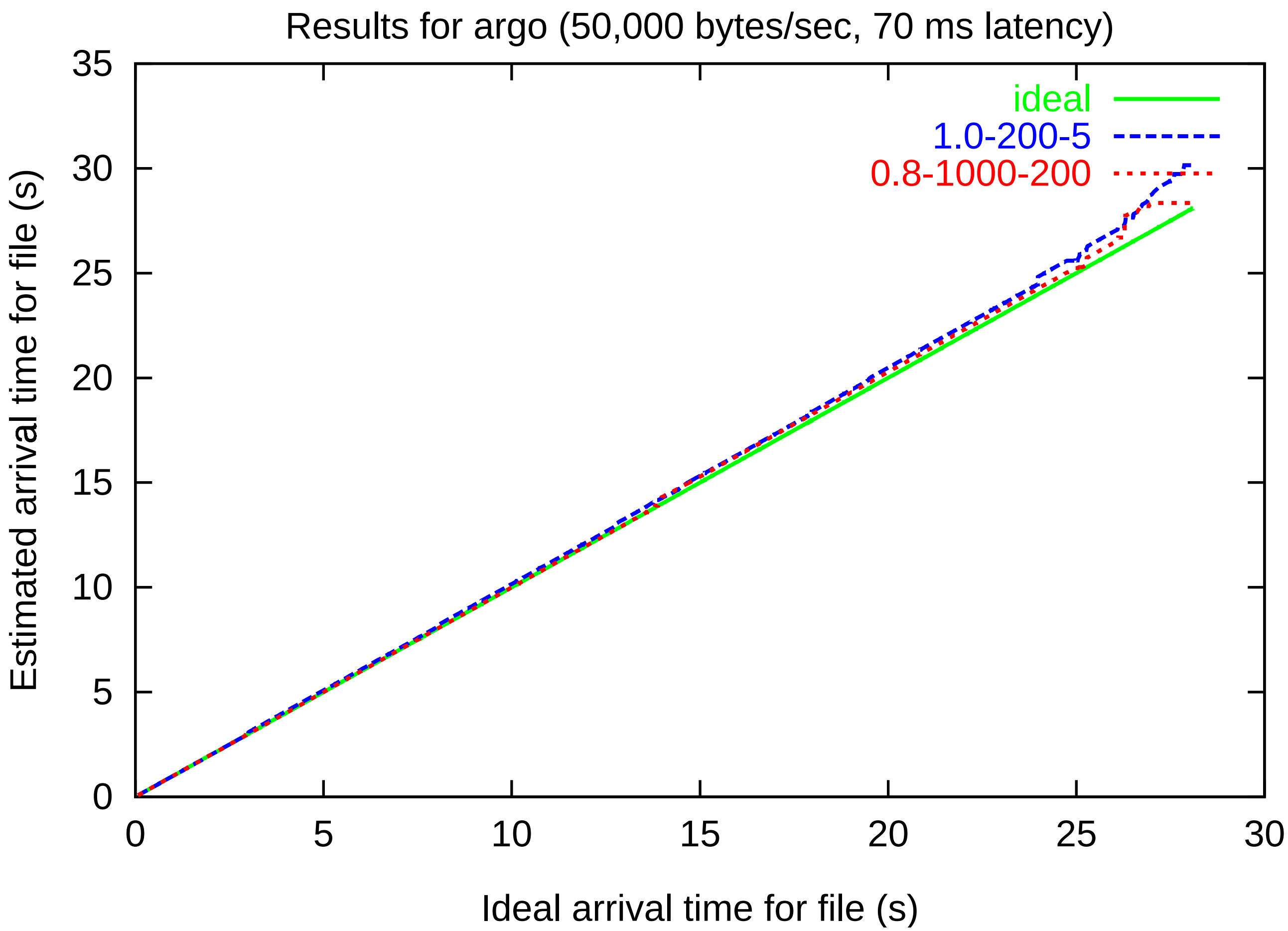
- Results for 50,000 bytes/second bandwidth, 70 milliseconds latency
- ‘Ideal’ bundling consists of 1 bundle containing all files needed, in correct order
- Looser bundling parameters help to reduce latency delays

Experiment 2

- A realistic application
- Bundlings generated from five profiles from Argo/UML
- Class and resource loading behavior very consistent
- Test done on input profile which was a member of the input set
- Note: the 'loose' bundlings (0.8-1000-200 and 0.8-1000-500) were identical for these input profiles

Experiment 2

Expected file arrival times vs. ideal for Argo/UML, 50,000 bytes/second bandwidth, 70 milliseconds latency

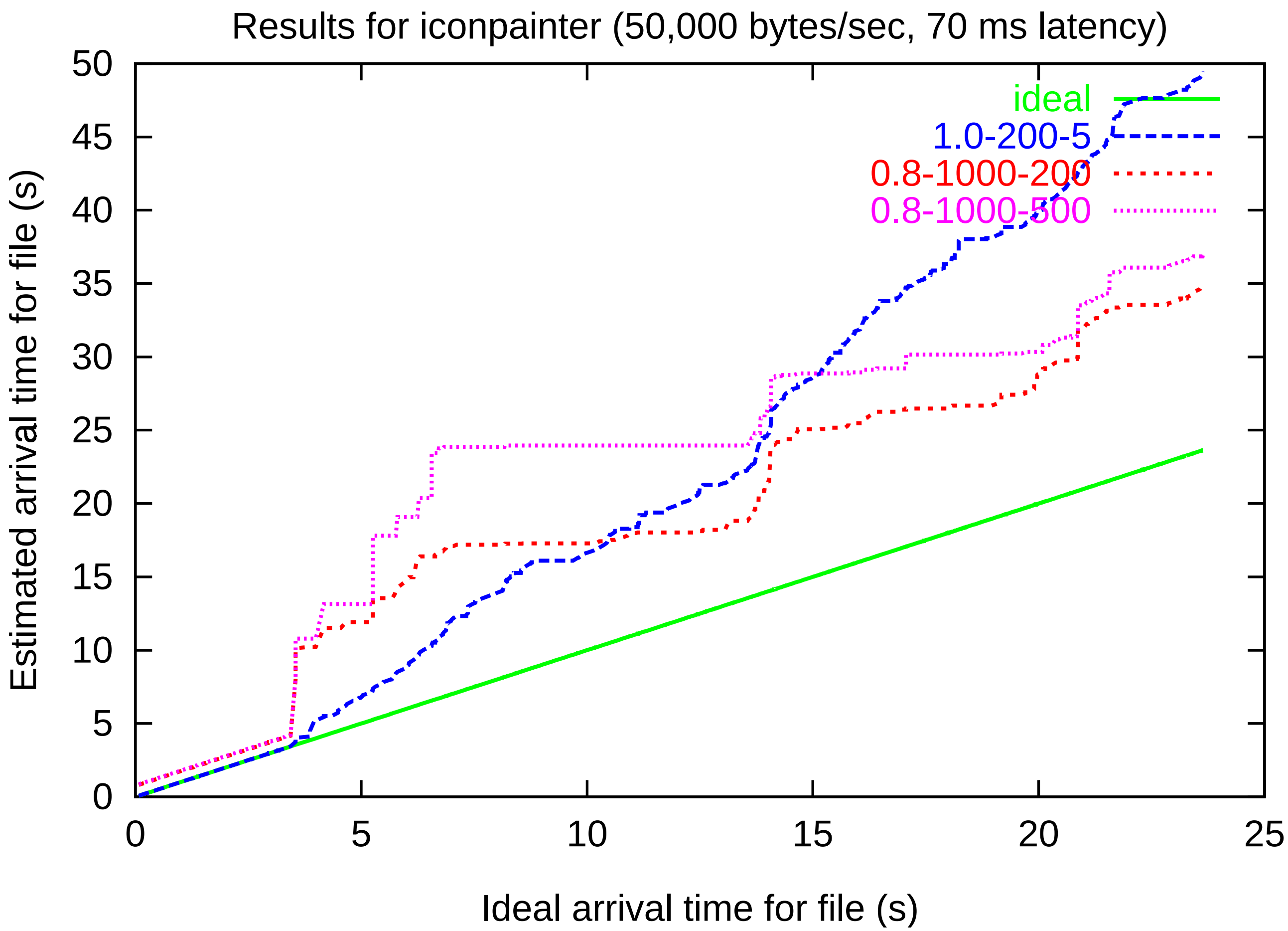


Experiment 3

- Test bundlings with applications not represented in input profiles
- To see how well bundlings perform when unexpected class and resource loading behavior is encountered
- Again, not a realistic application of bundling
- In a ‘real’ application, would want to continuously collect profiles and update bundlings correspondingly

Experiment 3

Expected file arrival times vs. ideal for IconPainter, 50,000 bytes/second bandwidth, 70 milliseconds latency



Outline

- Motivation
- Algorithm
- Implementation
- Experimental results
- Conclusions

Conclusions

- Archive formats may send files that are not needed
- Pure on-demand loading suffers too much from request latency
- Bundling is a compromise between archive and on-demand techniques
  - ▷ Can achieve desirable properties of both
  - ▷ Can be tuned for various network conditions (bandwidth, latency)