Dartmouth College

Dartmouth Digital Commons

Dartmouth College Undergraduate Theses

Theses and Dissertations

6-1-2002

Performance and Interoperability In Solar

A Abram White
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation

White, A Abram, "Performance and Interoperability In Solar" (2002). *Dartmouth College Undergraduate Theses*. 25.
https://digitalcommons.dartmouth.edu/senior_theses/25

Performance and Interoperability in Solar

A. Abram White
abewhite@mac.com

Senior Honors Thesis
Dartmouth College Computer Science
Advisor: David Kotz

Dartmouth College Technical Report TR2002-427

Abstract

Ubiquitous computing promises to integrate computers into our physical environment, surrounding us with applications that are able to adapt to our dynamics. Solar is a software infrastructure designed to deliver contextual information to these applications. To serve the large number and wide variety of context-aware devices envisioned by ubiquitous computing, Solar must exhibit both high performance and the ability to interoperate with many computing platforms. We created a testing framework to measure the performance of distributed systems such as Solar, as well as a pluggable data-transfer mechanism to support the dissemination of information to heterogeneous applications. This paper explores the testing framework developed, analyzes its findings concerning the performance of the current Solar prototype, presents several optimizations to Solar and their effects, and finally discusses the design of the pluggable data-transfer mechanism.

1   Introduction

The next generation of computing devices will be capable of adapting to our behavior and surroundings, becoming at the same time both more useful and less conspicuous than computers today. *Ubiquitous computing* is a vision of a world in which these "smart" devices pervade our environment; to date, however, there is very little software that is able to adapt its user's context.

The primary obstacle to the creation of context-aware applications may be the lack of a system capable of delivering the necessary contextual information [3]. A viable system must be able to disseminate diverse forms of context quickly, must be able to scale to handle a truly pervasive computing environment, and must be able to interoperate with the many heterogeneous devices in such an environment. Solar is a system infrastructure designed to meet these challenges.

This thesis explores measures taken to improve the performance and interoperability of Solar. It begins with a general introduction to ubiquitous computing and the Solar context-dissemination system. It then presents a testing framework we developed to measure the performance of highly distributed, highly configurable systems such as Solar. Next, it analyzes the performance of the Solar prototype using this testing framework. Following the analysis, it presents several optimizations we implemented,

along with their effects. Finally, it discusses the design of a mechanism we created to allow devices interacting with Solar to receive context data in their own preferred format, enabling Solar to operate in a heterogeneous computing environment.

2   Ubiquitous Computing

In the past, staying warm was an important daily concern, and the fire or stove often became the center of attention on cold winter nights. Today, thermostats and extensive heating systems are built into our homes, buildings, and cars, and though indoor heating is as important to our lives as ever, it rarely enters our thoughts. Advances in climate-control systems have allowed us to forget they even exist, enabling us to focus our attention on other matters.

The above example is meant to illustrate that technology often becomes most useful to us only when it effectively disappears from our consciousness. *Ubiquitous* (or *pervasive*) *computing* applies this principle to computers; it envisions a world in which computers are integral to our daily life, yet so seamlessly interwoven into the physical environment that they no longer occupy our attention [11]. While this vision may be compelling, it faces several challenges:

- o **Hardware.** To embed ubiquitous computing devices into our surroundings, the devices themselves must be small and inexpensive, but at the same time powerful enough to run the complex logic needed to adapt to changes in the environment. Rapid advances in hardware miniaturization and continuous exponential growth in computer processing power indicate that this challenge may be met within the next few years.

- o **Network.** The network is obviously critical in delivering contextual information to ubiquitous devices. The network infrastructure must be able to support heavy wireless traffic, must use standard protocols able to address large numbers of densely packed hosts, and must scale to potentially hundreds of devices *per room*. The growing prevalence of wireless networks, standardization on the TCP/IP protocols, and the emergence of IPv6 are all advances towards these ends.

- o **Software.** While computing hardware and the network infrastructure are well on their way to meeting the needs of ubiquitous computing, the necessary software is still in its infancy [7]. Two categories of software must be developed: "smart" applications, and a context dissemination system to support them.

  Traditional applications are not designed to cope with changes to the environment they run in. Smart applications, on the other hand, are programs designed to adapt to their user and setting. Their goal is to assist the user, and often to anticipate her needs, without getting in the way. The infamous Microsoft paperclip is an example of a failed smart application, and illustrates the delicate balance between being helpful and being a nuisance. There are many positive

examples of smart applications, however, such as location-aware Palm programs for displaying local entertainment options, stock price trackers that notify the user when certain criteria are met, or intelligent reminder systems that account for traffic and distance from the appointment location [9]. The common trend among these applications is that they are able to effectively use context to provide a service that adapts to changing conditions. Unfortunately, there is no standard framework for the dissemination of the location data, stock prices, traffic conditions, and other context these programs require. Each application must obtain the needed data in an *ad hoc* manner, and must handle difficult problems like mobility and variable network conditions on its own. The lack of a support infrastructure for context-aware applications is one of the major obstacles to ubiquitous computing.

2.1 Context Dissemination

A context-dissemination infrastructure must meet several requirements to be useful in a pervasive-computing environment:

- o **Flexibility.** The system must be able to deliver the diverse forms of data employed by various context-aware applications, and to communicate with these applications regardless of their native computing platform.

- o **Scalability.** The infrastructure must be able to deliver context information from thousands of sensors to thousands of applications employed by hundreds of users.

- o **Information Quality.** Context often comes from error-prone sensors and resource-constrained embedded devices. It is unreasonable to ask each application to transform this low-level, unreliable information into the high-level context it requires. The system must assist applications in ensuring a certain level of information quality.

- o **Mobility.** Both the sensors producing context data and the applications consuming it may be highly mobile. The infrastructure must continue to deliver the needed information during rapid location changes.

3 Solar

Solar is a system infrastructure designed to meet the challenges of supporting end-to-end information collection, dissemination, and utilization for adaptive ubiquitous applications [3]. As such, its primary goals correspond exactly to the challenges discussed in Section 2.1:

- o **Flexibility.** Solar must not restrict the content of context information, and it must be able to deliver this information to applications on heterogeneous computing

platforms.

- o **Scalability.** Solar must collect information from many sources and disseminate it to numerous clients without becoming a bottleneck in the information flow.

- o **Information Quality.** Solar must allow applications to access both raw sensor data and high-level refined context. Additionally, Solar should enable applications to easily manipulate context to best suit their needs.

- o **Mobility.** Solar should use location-dependent information sources to support mobile sources and applications. Solar must handle fast context changes as these devices travel across geographic space.

3.1 Design

Solar represents context as *events*. Sensors and other *sources* of context information publish streams of events, and applications subscribe to the streams they are interested in. The events themselves are objects, and the class of each event represents its type. Publishers are free to define their own event classes; thus Solar allows the dissemination of arbitrarily complex data.

Still, few applications want to work with the relatively low-level information published by most sources. Solar enables applications to construct new, higher-level event streams from existing streams through the use of *operators*. Operators are small, chainable programs that receive one or more event streams as input and publish a single event stream as output. Solar allows applications to reuse existing operators, or to define their own operators, which Solar will dynamically load and deploy across the network. Most operators perform one of four basic actions:

1. **Filter.** Filters are used to remove redundant or otherwise uninteresting events from a stream. For example, a sensor might publish the current temperature every 10 seconds, but the application might only want to receive temperature readings that exceed 90 degrees.

2. **Transform.** Transformation involves replacing events of one type with those of another. A location transformer might convert map coordinates to rooms within a building.

3. **Merge.** Merge operators subscribe to two or more event streams and simply re-publish all the events they receive as single stream. While not strictly necessary, merge operators can aid stream reuse, as we will see shortly.

4. **Aggregate.** Aggregators are the most complex form of operator. They output an event stream based on input from one or more streams, often in combination with internal state. An aggregator might be used to track the

price of a stock and publish an event only when the stock reaches a new high or low for the day.

Solar maintains a logical overview of event flow in the *operator graph*. The operator graph is an acyclic graph in which sources, operators, and applications are represented as nodes, and event streams as directed edges between them. Using the operator graph, Solar can identify overlapping event pathways so that these portions of the event flow can be shared by multiple applications. The reuse of event streams conserves network and computational resources and greatly enhances Solar's scalability.

To facilitate this reuse, Solar allows symbolic names to be attached to frequently used publishers. Applications can then subscribe to event flows using their names rather than having to specify the construction of the flow from the basic building blocks of sources and operators. Solar combines the symbolic names of publishers with the names of available *services* (printers, displays, etc.) and organizes them hierarchically to form the *naming tree*. The naming tree consists of both static names used to identify specific devices or users and context-sensitive names whose bindings may shift over time. For example, the static name /devices/printers/23 may refer to a specific printer (printer number 23) while the context-sensitive name /labs/005/devices refers to a dynamic list of devices (possibly including printer 23) that is calculated based on the current content of lab 005, as determined by location sensor data. Context-sensitive names can adapt to mobile sources in a manner that is completely transparent to applications.
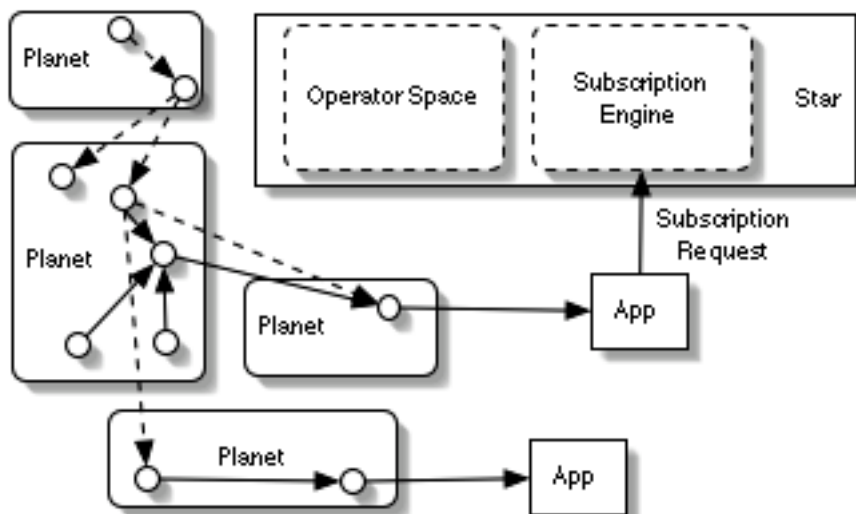
3.2   Implementation

Figure 1: The architecture of the Solar system. Circles represent sources and operators executing on Planets. The dashed arrows are tree edges in the name space while the solid arrows represent subscription links in the operator graph [3].

The Solar prototype is implemented in Java. At the center of the Solar system is the Star, which runs in its own Java Virtual Machine (JVM). The Star maintains the operator graph and naming tree, and receives subscription requests from applications.

Applications lie outside the Solar system; subscription requests are specified using a simple XML subscription language, and are sent to the Star using a client-side Solar library. When a subscription requires the deployment of a new operator, the Star instantiates the operator's object on one of many Planets. A Planet runs within a JVM and acts as an execution platform for Solar sources and operators. Planets are typically distributed throughout the local network.

Solar event types are represented as Java classes, and each event is a Java object. Events are transferred between Planets and from Planets to applications using standard Java serialization.

4   Performance Testing Framework

Studying the design and implementation of Solar shows us that it meets the challenges of flexibility, information quality, and mobility posed to context dissemination systems in section 2.1. The only way to measure the performance and scalability of Solar, and therefore its viability as an infrastructure for truly pervasive computing, though, is to gather empirical data. Unfortunately, Solar exhibits many characteristics that make it unsuitable for evaluation using traditional performance-testing software:

- o **Distributed.** Solar is a highly distributed system. Any realistic tests of Solar's performance and scalability must incorporate multiple Planets running on several physical hosts.

- o **Server.** Solar Stars and Planets are independent servers that are meant to be run as standalone programs. They do not expose APIs that allow them to be invoked as components from within other applications.

- o **Highly Configurable.** There are no "standard" configurations in Solar; instead, each application specifies its own tree of sources and operators. To accurately simulate these applications in testing scenarios, each test case must be free to create arbitrary operator graphs using arbitrary network hosts. Additionally, the types of results from test cases may vary depending on the type of test performed.

- o **Variables.** Because Solar is a complex system, each test of Solar may involve many runs using different combinations of multiple variables. For example, a test measuring event throughput using filter operators might vary the number of clients, the event size, and the fraction of events filtered.

- o **Evolving.** Solar is an evolving system. Testing code placed within the Solar code-base is likely to break as Solar develops over time. Individual test cases may rely on Solar's public APIs to interact with the system, but the testing framework itself must lie outside of Solar and be general enough to continue to function as Solar changes.

We decided to develop our own performance testing framework to meet the needs of complex, highly distributed systems like Solar. In addition to supporting the characteristics of such systems listed above, we set the following goals for our framework:

- **Simplicity.** While the systems our framework is designed to test are complex, testing them should not be. Implementing test cases should not involve an arduous integration process to tie into the testing framework. Furthermore, despite the highly configurable nature of systems like Solar, setting up and describing test runs to the framework should be made as easy as possible.

- **Automation.** Tests may take minutes, hours, or even days to run. The testing framework should be able to process tests without constant developer interaction.

- **Fully Documented Results.** It is all too easy to painstakingly collect volumes of results, only to later discover that we no longer remember the tests we ran to get them. To overcome this problem, results outputted by the testing framework should include the complete test configuration.

- **Multiple Result Formats.** It should be possible to easily transform test results into multiple formats, such as graphs or tables.
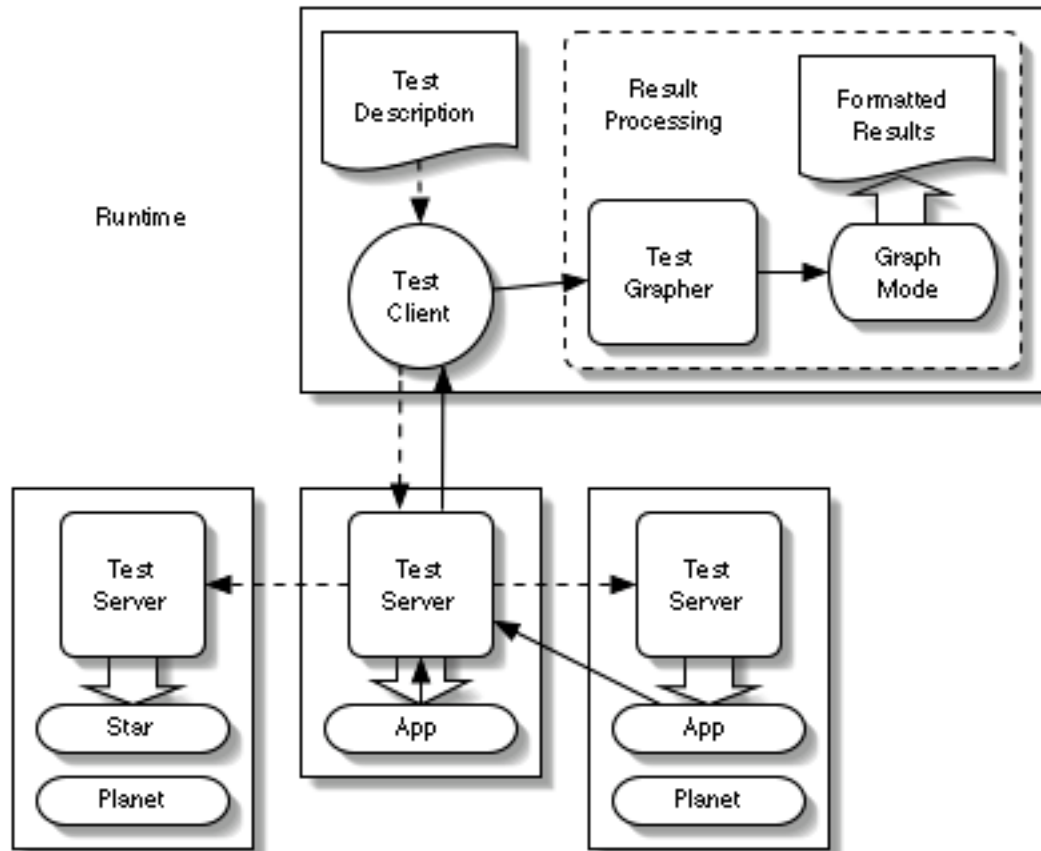
4.1 Design

Figure 2: The architecture of the testing framework. The dashed arrows represent test descriptions. The solid arrows represent test results. A wide arrow from A to B indicates that A spawns or produces B.

The testing framework is divided into two primary components: the runtime system and the result processing system. We focus on each of these parts in turn.

4.1.1 Runtime System

The runtime component of the testing framework is centered on the *test server*. Test servers are multithreaded servers responsible for running all or part of a test. The test server that receives the request to run a particular test oversees the test's execution and collects its results. Most tests involve running applications on multiple hosts; the overseeing test server also coordinates with other test servers running on these hosts to distribute applications according to the test specifications. Test servers communicate with each other and with other applications through the use of *command* objects, where each command type contains the logic for executing that command. This design enables us to easily extend the set of commands a server understands without disrupting its core API.

Test servers receive requests from the user through a *test client*. A test client translates the user's request into the server's native test format, sends the test to the server, receives the results from the server, and then translates these results into a format suitable for display back to the user. Different test clients can accept different forms of input and produce different forms of output. We developed one test client for use within Java applications; this client inputs and outputs Java objects. A second test client we developed is meant for command-line use; it accepts XML documents describing tests to run and produces XML documents containing the results.

Test servers represent tests internally as Java objects. These objects have several responsibilities within the framework:

1. They hold global configuration information pertaining to the test.

2. They list the variables for the test, along with enumerations or start/stop/step rules for the variables' possible values.

3. During testing, test objects act as iterators over the possible values their variables can take. The test server runs the test for each possible combination of variable values. We employ a recursive algorithm to evaluate the possible variable value combinations as if the variables are placed in nested loops.

4. They act as containers for the server objects the test uses; server objects are covered below.

5. They acts as containers for their own results. Once a test has been run for all possible combinations of variable values, the test server processes the results it has collected; for numeric results it calculates simple statistics like min, max, mean, and standard deviation. The results and statistics are then stored within the test object itself, and the test object is transmitted back to the waiting test client. The client can obtain the results from the test object, together with the original test configuration data; thus the result is fully documented.

Tests are often placed together in *test groups*. Test groups are little more than conglomerations of related tests. Test groups can also hold arbitrary name/value pairs used to record information about the testing environment, such as the operating system used, the network speed, etc. For convenience, test servers and test clients have overloaded APIs for handling entire test groups as well as individual tests.

As we noted above, tests contain one or more *server* objects. Each server object is the logical representation of a physical host where a portion of the test should be run. The server object lists the set of applications that should be executed on the machine it represents. It also holds configuration information that is shared by these applications.

The testing framework supports the execution of three types of applications: Stars, Planets, and user-defined programs. User-defined programs are specified in the test

description by naming the Java class that acts as the entry point into the program. To make integration with the testing framework as painless as possible, there is only one requirement placed on this class: it must implement the standard Java `Runnable` interface, which is made up of a single `run()` method. The system invokes this method and waits for it to complete. If the program is meant to produce a test result, it can implement the `TestCase` interface, which extends the `Runnable` interface with a `getResult()` method. The testing framework automatically detects when applications have results to return, and sends these results back to the overseeing test server for processing.

Stars, Planets, and user-defined applications are all run within separate processes. The test server spawns these processes at the beginning of each test run, and kills them once the run has completed. This ensures that test runs are completely independent from each other, and that the server process does not interfere with the execution of applications.

We have mentioned application configuration several times: tests can contain configuration information global to all applications, servers can contain configuration information shared by the applications on that server, and application representations can contain configuration information for that particular application. Each configuration parameter is specified as a pair of strings representing a property name and the value that property should take. When an application instance is constructed, the testing framework attempts to match the supplied property names with the available *setter* methods of the instance. If a match is found, the framework converts the property value to the type expected by the method and then invokes the method with the converted value. For example, a user-defined throughput application might have a method `setStarHost(String host, int port)` to tell it where to send its subscription request. The configuration for this application could ensure that this method is invoked with the host name "agent1.cs.dartmouth.edu" and the port number 8080 by specifying that the property "starHost" has the value "agent1.cs.dartmouth.edu,8080". Thus the chore of configuration is entirely handled by the testing framework.

4.1.2   Result Processing System

On a test's conclusion, its results are packed into the test object and sent back to the client. What format should the client use to display these results to the user? There is no single answer to this question; the correct format depends on the type of test and the user's needs. Therefore, we designed a dynamic result-processing system that includes a plug-in architecture for creating new output styles.

In the result processing system, the *test grapher* is responsible for accepting test results as input, along with options for how the results should be processed. One of these options specifies the *graph mode* the test grapher uses to format results. Graph modes are system plug-ins for transforming result data. Developers can create their own graph modes to format this data in any style they choose, and, despite the names of these components, not all formats involve graphs. In fact, though one of the graph modes we developed does in

fact produce line graphs in Portable Network Graphics (PNG) format, another outputs a simple text file of comma-separated values.

4.2   Example

To make the preceding discussion of the testing framework more concrete, it may be helpful to consider an example. Assume that we have created a simple Solar source that outputs a series of events as quickly as possible. Additionally, we have created a Solar application called *SolarThroughput* that subscribes to this source and measures the amount of time it takes to receive all of the events, then calculates the event throughput in events per second.

```xml
<?xml version="1.0" standalone="yes"?>
<test-group name="Throughput (2 Receivers)">

   <!-- test description -->
   <test name="Solar">
      <server host="agentc24.cs.dartmouth.edu">
         <star />
      </server>
      <server host="agentc23.cs.dartmouth.edu">
         <planet />
      </server>
      <server host="agentc22.cs.dartmouth.edu">
         <app name="throughput1" class="solar.test.cases.SolarThroughput"/>
      </server>
      <server host="agentc21.cs.dartmouth.edu">
         <app name="throughput2" class="solar.test.cases.SolarThroughput"/>
      </server>

      <!-- vary the payload of the message events -->
      <var name="randomInstance.stringLength"
         start="0" end="1000" step="100"/>

      <!-- global configuration -->
      <param name="iterations" value="1000"/>
      <param name="starHost" value="agentc24.cs.dartmouth.edu"/>
   </test>

   <!-- used in graphing -->
   <param name="categoryAxisLabel" value="Message Length (characters)"/>
   <param name="valueAxisLabel" value="Throughput (messages / second)"/>

   <!-- env properties -->
   <param name="date" value="2002/2/25"/>
   <param name="os" value="RedHat 7.1"/>
   <param name="network-speed" value="100Mbs"/>
   <param name="java-version" value="1.4.0 FC"/>
</test-group>
```
Figure 3: Test description suitable for use with the XML test client.

To run this test case, we must first create the test description. Figure 3 depicts a description of the test suitable for use with the XML test client. It is not necessary to understand the details of this document, but the gist of it should be clear. The test uses four servers: one to run the Star, one to run the Planet for the event source, and the others to run two instances of the throughput application. The event type the test uses is the

11

MessageEvent, a standard Solar event type whose payload consists of a string message. The test uses a variable to systematically vary the size of this message, and by extension the network size of the transmitted events.

Once the test description is written, we must start test servers on each of the hosts mentioned in the test. We then use the XML test client to transfer our XML description as a test object to one of the servers. This server divides the test object into its constituent hosts and distributes the application-description objects to the relevant test servers. Each server configures the described applications with the supplied parameters, and then spawns the applications as separate processes. Once the results from the two throughput applications have been sent back to the coordinating test server, the processes are killed.

Next, the test server asks the test object to step to its next variable value, and the entire processes repeats. Eventually the variable values are exhausted. At this point the test server performs some final processing on the results, packs them into the original test object, and transfers it back to the waiting XML client. The client outputs the test object, including the results it now contains, as an XML document. Figure 4 displays a result document from a typical run of our test.

```
<test-group name="Throughput (2 Receivers)">
    <test name="Solar">
        <server host="agentc24.cs.dartmouth.edu" port="8108">
            <star data-port="4104" control-port="5105"/>
        </server>
        <server host="agentc23.cs.dartmouth.edu" port="8108">
            <planet data-port="6106" control-port="7107"/>
        </server>

        remainder of server configuration omitted…

        <result min="290.697" max="294.811" mean="292.754" std-dev="2.056">
            <var name="randomInstance.stringLength" value="0.0"/>
            <value app-name="throughput1" error="false" value="290.697"/>
            <value app-name="throughput2" error="false" value="294.801"/>
        </result>
        <result min="291.460" max="292.740" mean="292.100" std-dev="0.639">
            <var name="randomInstance.stringLength" value="100.0"/>
            <value app-name="throughput1" error="false" value="291.460"/>
            <value app-name="throughput2" error="false" value="292.740"/>
        </result>

        remainder of results omitted…

    </test>

    global parameters omitted…

</test-group>
```

Figure 4: Test results from the XML test client.

Finally, these results can be passed through the test grapher for formatting. The test grapher supports both XML, which it translates into test objects, and test objects

themselves as input. Figure 5 is an image produced by the test grapher using the PNG graph mode.
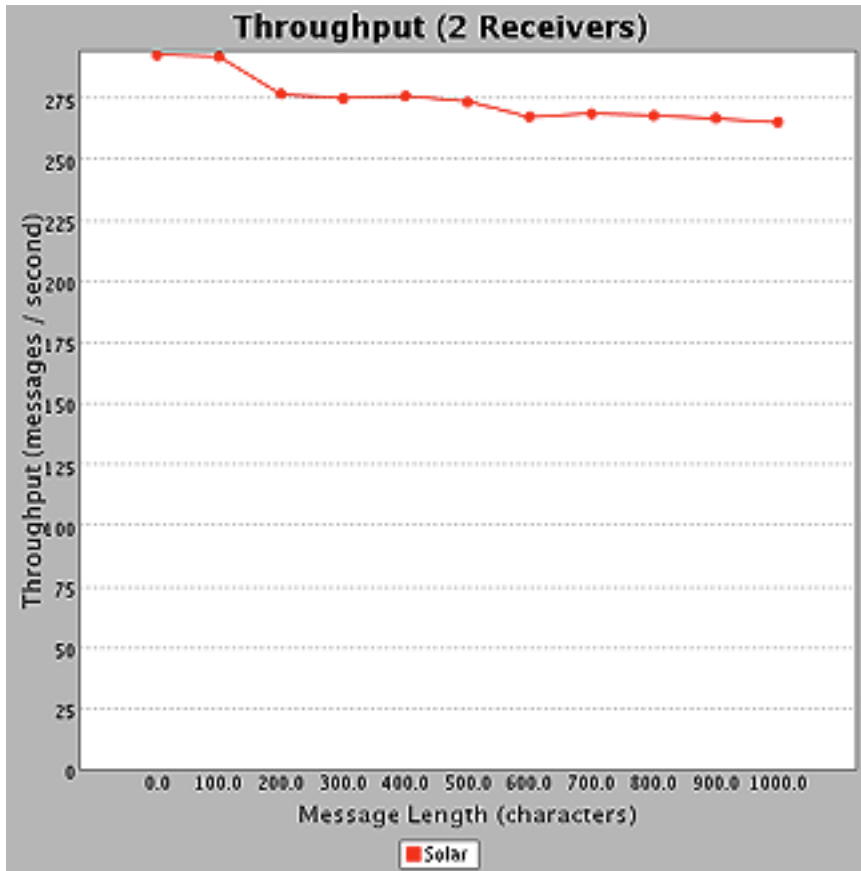

Figure 5: Output from the test grapher using the PNG graph mode.

5   Test Cases

With a suitable testing framework in hand, we turned our attention to gathering data on Solar's performance and scalability. Solar is designed to support a wide variety of context-sensitive applications, and it would be impossible to evaluate the system's performance in every possible scenario. Instead, we focused our attention on the following aspects of context flow:

- o   **Latency.** How much time does it take to transfer an event from its source to its final destination?

- o   **Throughput.** Can Solar deliver a series of rapid-fire events without becoming a bottleneck? How does Solar scale when delivering these events to multiple receivers?

- o **Filter.** Most applications will probably only be interested in a small fraction of the context information that is potentially available to them. How does reducing the event flow using filters influence performance?

5.1 Design

For each of the facets of performance listed above, we wanted to compare Solar to other methods of context dissemination designed specifically to maximize performance, often at the expense of functionality. This comparison would give us a baseline measure of how much overhead Solar incurs due to its extreme flexibility. Thus, for each test, we wrote test cases designed to run through Solar as well as cases designed to run standalone.

To simplify the development and use of our test cases, we gave them a core set of common configuration options:

- o **Data Type.** The type of raw data or event to use within the test.

- o **Data Size.** The size of each piece of data or of the payload of each event.

- o **Iterations.** The number of iterations to average the test results over. The exact meaning of this parameter changes with the test.

- o **Transfer Mode.** The method to use in sending the context data. In Solar tests, this parameter is ignored, because Solar takes care of event transfer using Java serialization. In standalone tests, however, other data-transfer methods are available:
  - o **Byte Stream.** Raw bytes are sent through Java sockets directly.
  - o **String Stream.** This transfer method uses character streams in place of bytes.
  - o **Serialized String Stream.** Like the string stream, this method also transfers strings, but employs Java serialization rather than character streams.
  - o **Serialized Object Stream.** Solar events are transmitted using Java serialization. This method is closest to the event transmission process of Solar.

5.1.1 Latency

The latency test is designed to measure the cost of routing context through Solar, as opposed to sending it straight from the data source to the waiting receiver. This latter case is easily modeled by developing small standalone client and receiver programs. A data packet of the configured type and size is sent from the client to the receiver, which

bounces the packet back to the client. The round-trip transfer time is a measure of system latency. Performing a complete round trip rather than sending data one way enables the client to have total control over the timing of the transfer, which avoids clock synchronization issues that arise when multiple machines must coordinate to time a process. Events are bounced back and forth like this for the number of iterations given in the test configuration, and the average round trip time is reported as the test result.
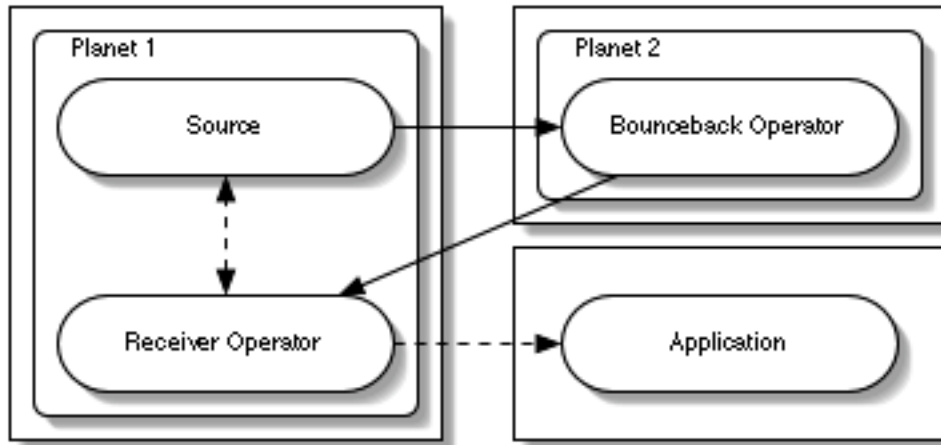

Figure 6: Solar version of the latency test. The solid arrows indicate data event flow. The dashed arrows denote timing data.

The Solar version of this test is diagrammed in Figure 6. An event containing a payload of the specified size is sent from Planet 1 to a waiting bounce-back operator on Planet 2. The bounce-back operator receives the event and re-publishes it to a receiver operator back on Planet 1, mimicking the round trip performed in the standalone test case. The receiver operator coordinates with the source to time the trip; because the receiver and source are on the same Planet, clock synchronization issues are once again avoided. The computed time is then sent as a special event type to the test application. As in the standalone case, this process is repeated for the specified number of iterations, and the final latency calculated as an average over the round trip times observed.

In each test, we varied the size of the data packet or event payload to assess its influence on transfer times.

5.1.2 Throughput

Throughput is a measure of how much data can be passed through a system per unit time. As an infrastructure designed to support ubiquitous computing environments consisting of thousands of devices and hundreds of users, Solar must support high throughput of context information. Additionally, this throughput should not degrade as the system scales to more and more users.

The standalone version of the throughput test consists of a lone sender program and a configurable number of receiver programs. The sender transmits a single start event to each receiver, then fires them a stream of data packets of the specified type and size.

Each packet is sent in round-robin fashion to each receiver; the number of packets sent is determined by the *iterations* parameter of the test. The receivers measure the amount of time that elapses between the arrival of the start event and the arrival of the last data packet, and then divide the total number of packets received by this time to calculate the throughput in packets per second.

The Solar version of the test is similar. Each application in the test subscribes to the same source, which transmits a single start event and then a stream of data events whose payload is the configured size. Solar handles the distribution of events to each subscribed application.

As in the latency tests, we varied the size of the data packet or event payload across test runs.

5.1.3 Filter

Solar filters have no clear analog in simple sender/receiver systems; therefore, we did not implement a standalone filter test. The Solar version of the filter test is straightforward: the test application subscribes to a filter operator, which in turn subscribes to a source that fires a stream of events of the specified type and size, just as in the throughput test. The filter operator, though, is configured to remove a fraction of the events from the stream. The placement of the filter can be toggled between three settings, as shown in Figure 7:

1. **Source.** The filter is placed within the source program, so that events that would have been filtered are instead simply never published. No operators are used.

2. **Independent Host.** The filter is an operator, placed on a separate host from the source and from the receiving application, so that events must take an extra hop through the filter's Planet.

3. **Application.** Filtering is performed in the application program. In this setting 100% of the events are transferred from source to receiver, just as in the throughput test without filters. No operators are used.
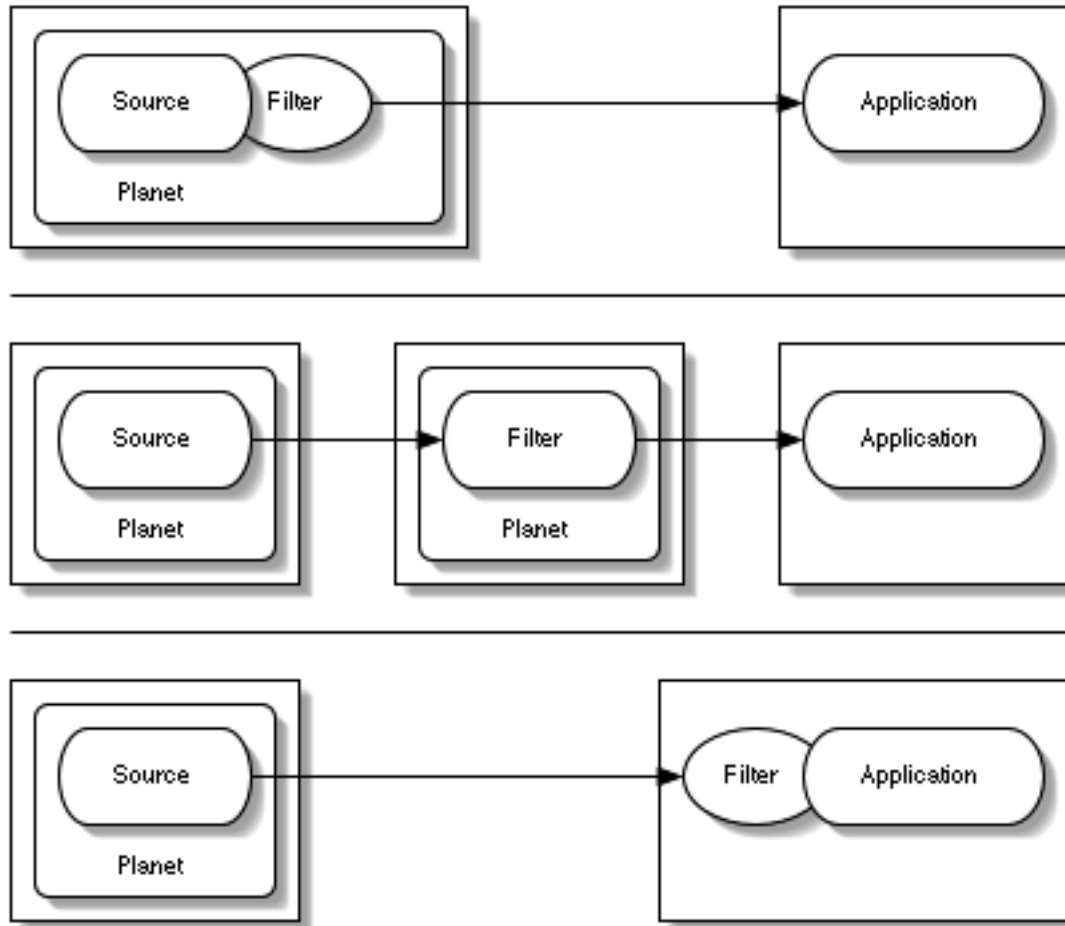
Figure 7: The three possible placements of the filter operator in the Solar filter test.

5.2 Profiler

Profilers are tools for analyzing the resource consumption of various components in a program. CPU time, memory usage, and other metrics are broken down by method and thread. Testing frameworks like ours and profilers compliment each other: testing frameworks can assess the overall performance of a system, but cannot shed any light onto the reasons for this performance. Profilers, on the other hand, analyze the inner workings of the program and determine what pieces might be slowing it down; however, they cannot measure total performance.

To get a more complete view of Solar's performance, we supplemented our testing framework cases with profiling data from the JProbe Java profiler [6].

6 Findings

This section presents results and analysis of the tests described in Section 5. We executed all tests on 450Mhz Pentium II workstations with 256MB RAM running RedHat Linnux 7.1 and the final candidate release of the Java Development Kit version

1.4. We ran multi-host tests on a cluster of these workstations connected by an isolated 100Mbps switched network.

6.1 Latency

Figure 8 presents the results of the latency tests. Solar was much slower than the standalone methods of data transfer in context delivery speed, though the difference is less apparent in the standalone case that also relied on event serialization. Curiously, the transfer mode using serialization on strings performed better than the mode utilizing character streams; we believe that this is due to the buffering that Java serialization employs. Also, because strings are a native Java type, the serialization process for them has been extensively optimized.

Aside from this minor anomaly, the results are not surprising: the transfer modes employing the simplest data conversion and the smallest transmission size exhibited the lowest latency. This implies that decreasing the network transfer size of events and replacing Java serialization with a less complex mechanism may reduce latency in Solar.
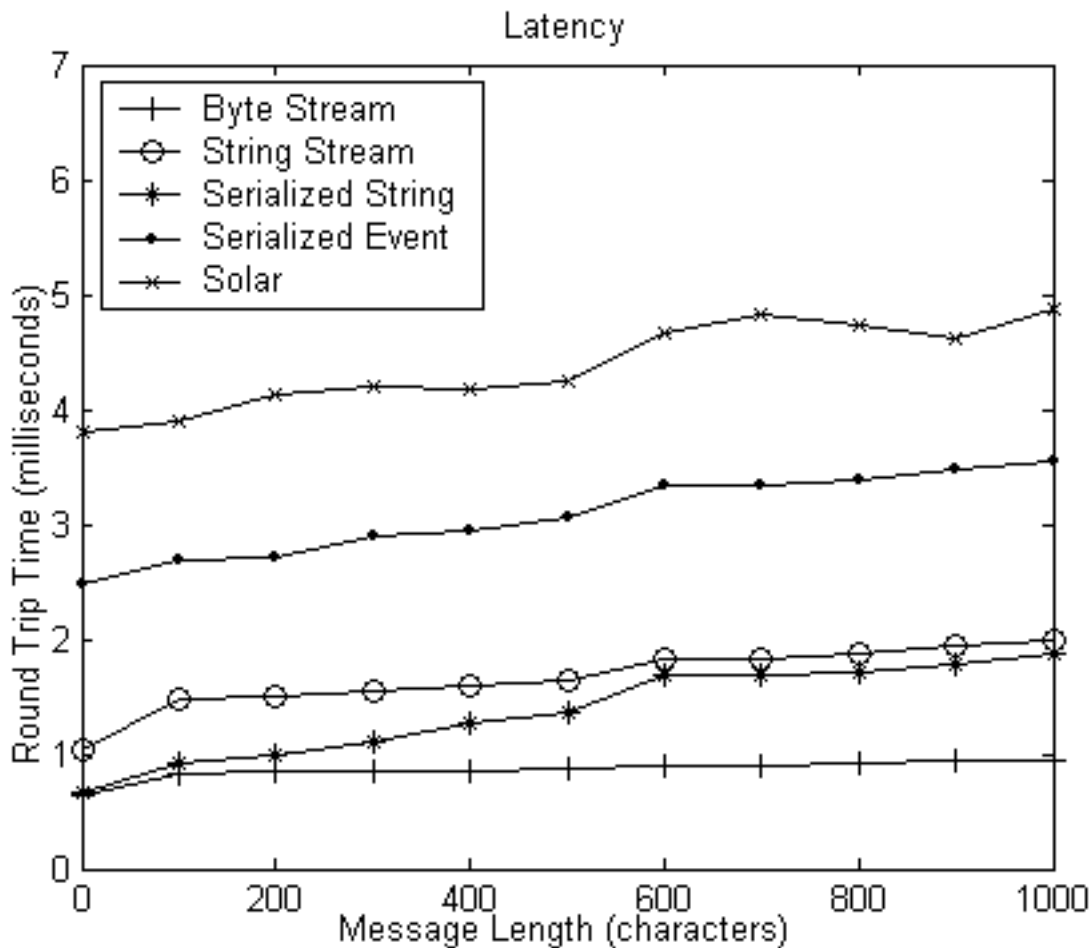

Figure 8: Results of the latency tests.

6.2   Throughput

We conducted throughput tests with both one and 10 receivers. The results for a single receiver are shown in Figure 9; overall they are quite similar to the results of the latency test discussed in the preceding section (except that in throughput graphs, higher is better). Transfer modes that minimize network traffic and require little or no work to prepare data for transmission dominated, though even the standalone test case that shares Solar's mechanism of event serialization bested Solar by up to 75%. This result implies that other inefficiencies in Solar are also contributing to its low throughput.
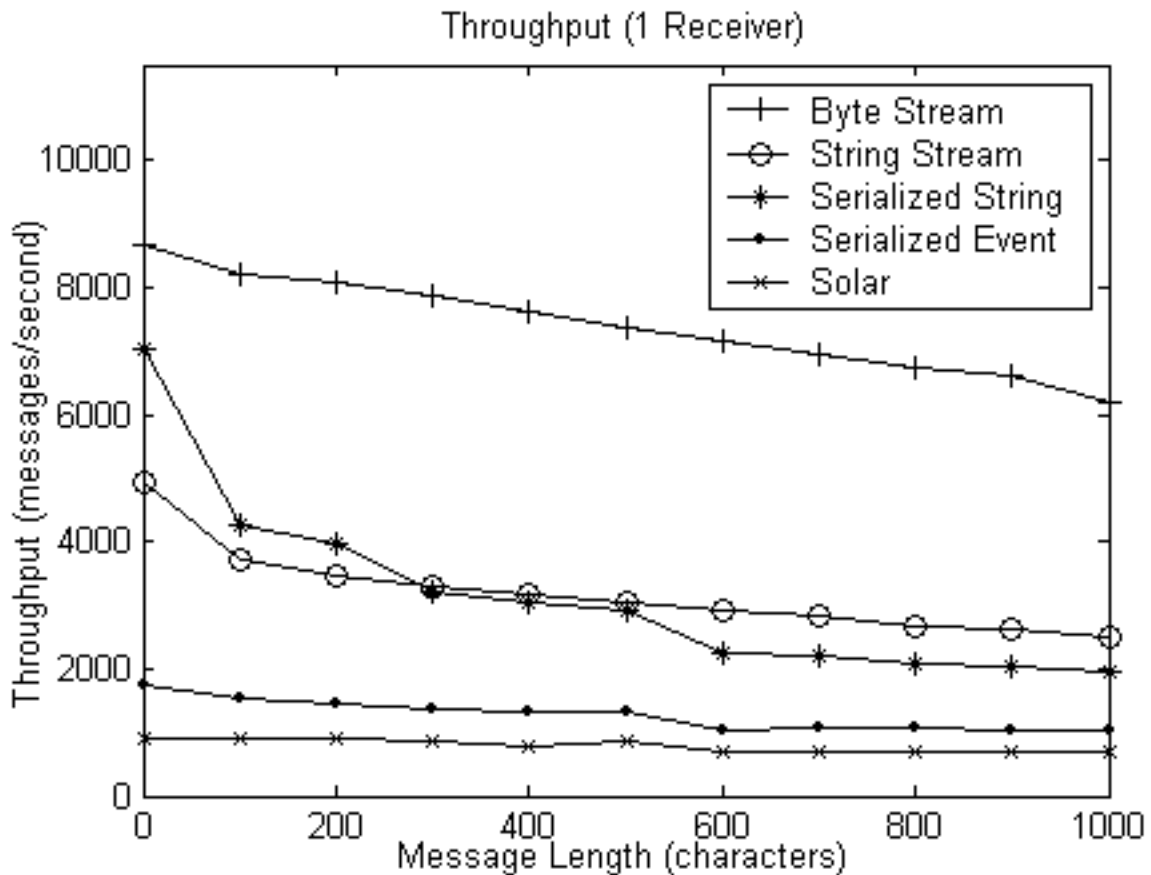

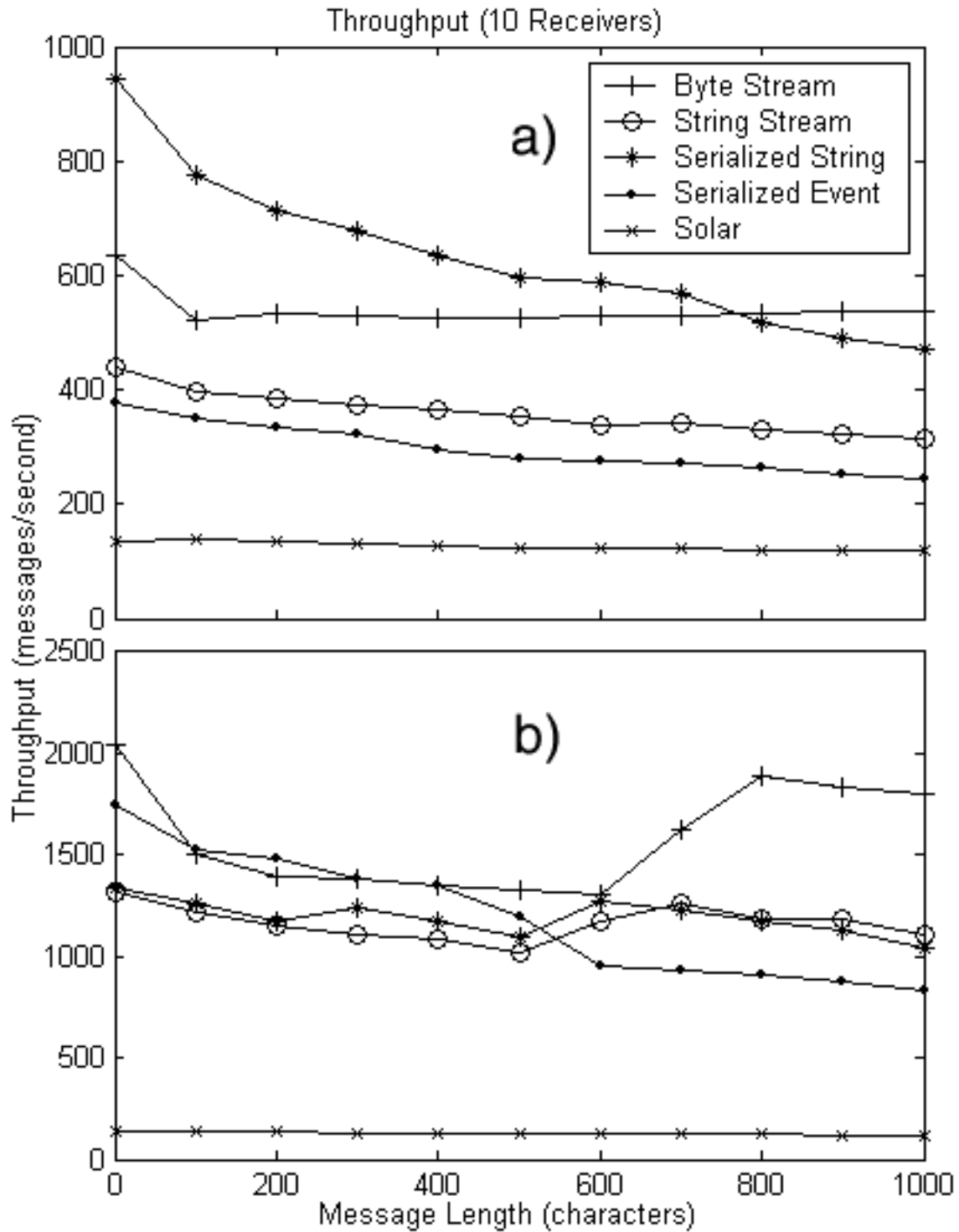Figure 9: Results of throughput tests with one receiver.

Figure 10: Results of throughput tests with 10 receivers: a) the test cases were run without shared serialization; b) shared serialization was enabled for the standalone versions of the test, leading to the apparent increase in performance for these cases.

We ran the 10-receiver throughput tests to evaluate Solar's scalability. Figure 10 plots the throughput *per receiver* against the message length. Comparing the single receiver results from Figure 9 with the base case of 10 receivers displayed in Figure 10a, it is easy

to see that Solar's throughput *per receiver* degrades more or less linearly with the number of subscribers; Solar's total throughput remains constant. The standalone tests initially exhibit similar behavior; however, enabling *shared serialization* in the standalone test cases boosts their performance and scalability tremendously.

Normally, data bound for multiple recipients is re-packaged for each transmission. Shared serialization is an optimization in which data is serialized once into a network-ready form, and then sent to each client without further modification. In addition to reducing the computation time involved in distributed data transfer, shared serialization reduces the gap between fast and slow data-transfer modes, as the clustered results in Figure 10b attest. This result indicates that Solar, which is forced to use a heavyweight serialization mechanism to handle arbitrarily complex context, could benefit from the implementation of shared event serialization.

6.3   Filter

Figure 11 plots the average time for each of 10 subscribers to receive a stream of events against the fraction of events filtered under various filter placements. Of the tests we ran, this one was designed to most closely approximate real-world usage, where the vast majority of available context will be ignored. Of the tests we ran, this one also exhibited the most encouraging results.

There are two reasons why the results of this test are encouraging. First, the results demonstrate that the operator-level filtering employed by Solar drastically increased performance over application-level filtering. When 90% of the 1000 streamed events were filtered, the application using Solar's operator filtering was a full 5 times faster than the application that executed filtering locally.

Second, the test results indicate that filtering through operators is efficient, and that filter operators do not have to be deployed to the same physical host as the event source to realize performance gains. In fact, filtering within the source program itself exhibits only slightly better performance than filtering using an operator on an independent server, though the gap may widen as network bandwidth decreases. Thus Solar does not have to lock itself in to a specific operator placement for the sake of efficiency.
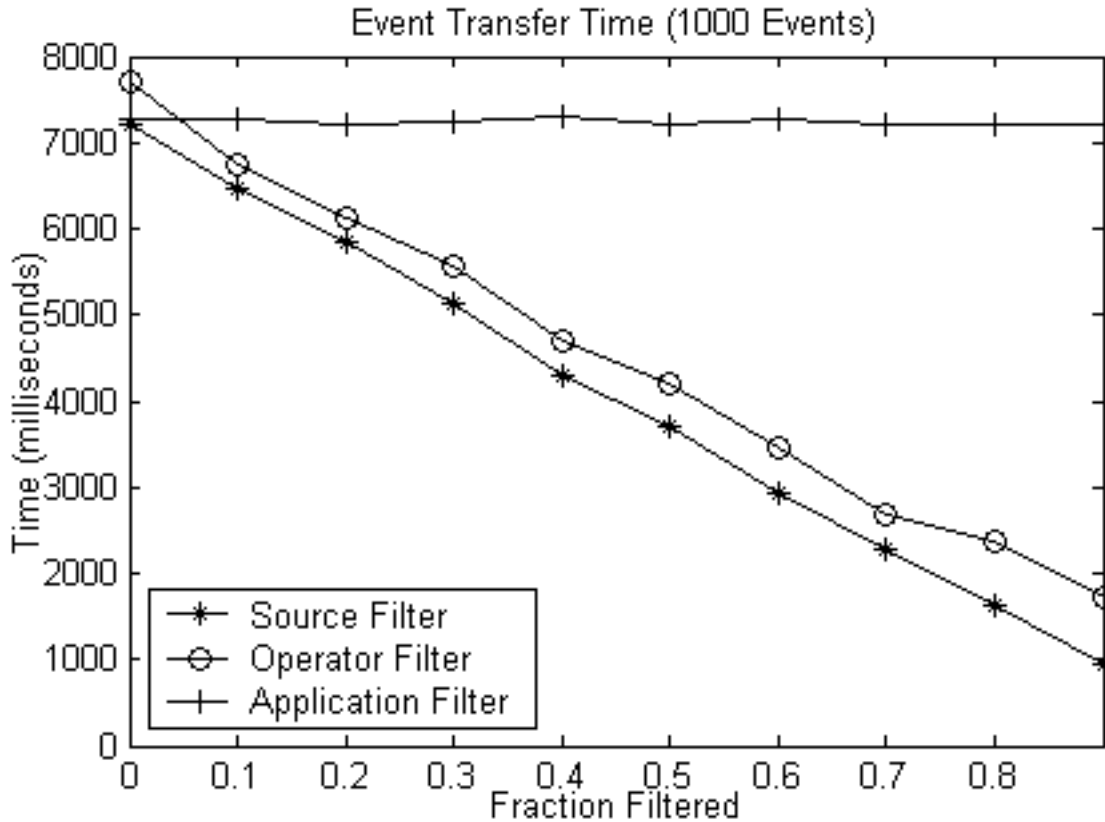
Figure 11: Results of filter tests in which a source with 10 subscribers publishes 1000 events. The graph plots the average time for each subscriber to receive all expected events vs. the fraction of events that were filtered.

6.4 Profiler

The results of the latency and throughput tests indicate that while complex serialization and large event size are at least partially responsible for Solar's sluggishness, there may be other factors involved as well. For a hint as to what else might be preventing Solar from maximizing its potential performance, we turned to JProbe [6], a Java profiler.

We ran JProbe on a Solar Planet during a throughput test. While we have not yet finished analyzing JProbe's output, one piece of information immediately caught our attention: Solar spends a large amount of time executing event-queuing methods.

Solar relies heavily on threading and queuing to dispatch events. The profiler data suggest that optimizing these aspects of the system could result in significant performance gains; however, further investigation is required.

7   Optimizations

Based on the test results presented in the preceding section, we implemented several optimizations intended primarily to improve Solar's throughput. The first such optimization was to decrease the size of Solar events. By simplifying some of the standard information included in each event, we were able to achieve a constant 222-byte decrease in the network transfer size of events without any loss of flexibility or additional restrictions on event content. Because most events tend to contain very small payloads, such as temperature readings or single stock prices, 222 bytes can amount to a significant fraction of the total event size (30% or more). Furthermore, these numbers assume the use of Java serialization as the network-transfer mechanism. Java serialization is a compact format; other more verbose formats will experience an even greater drop in total network size.

After reducing the size of events, we wanted to decrease the amount of time spent preparing them for network transfer. The standalone throughput tests showed large performance gains when data bound for multiple receivers was serialized once up-front, and then transmitted in serialized form to each client, as opposed to re-serializing the data for each intended destination. Thus, we implemented a similar system for Solar events published to multiple subscribers.

Unfortunately, Solar's execution of shared serialization cannot be as simplistic as the approach taken in the standalone tests, and therefore the performance gains realized are less dramatic. Two factors contribute to the added complexity of the Solar implementation:

1. Each Solar event must be annotated with its final destination. Therefore, while the event itself can be serialized once for a large number of clients, the destination information, which is clearly unique to each client, must still be serialized independently.

2. Different receivers can require different formats for transferred events (this functionality is covered in Section 8). The algorithm implementing shared serialization must group receivers by their preferred format, then serialize the context data once *per group* to ensure that each receiver is sent the event in the expected mode.

Despite these difficulties, we were able to increase throughput by approximately 85% in test cases involving 10 receivers, assuming that they use a common result format. Figure 12 compares the performance of Solar before and after the realization of shared serialization.
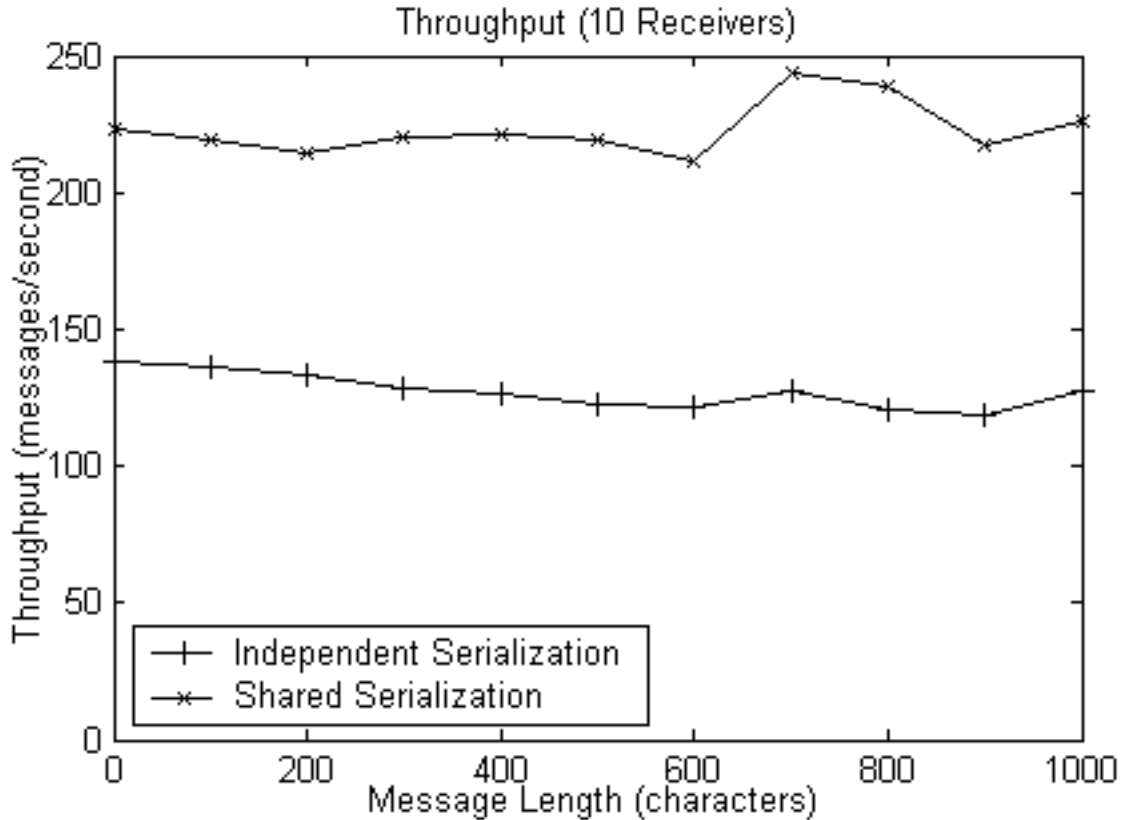
Figure 12: Performance gains using shared serialization for a throughput test involving 10 receivers.

The final optimization we attempted takes advantage of the pluggable data transfer mechanism we developed to increase Solar's interoperability. The details of this mechanism are explored in Section 8; suffice it to say that Solar is no longer limited to Java serialization for event transmission.

We created two new transfer modes designed to reduce network traffic and serialization time: the *command mode*, and the *data-stream mode*. The command mode serializes the internal events Solar uses for communication between Stars and Planets. The only important piece of information in these events is the command they denote, which is identified by a string. The command mode extracts this string and sends it over the network, then reconstructs the proper event on the receiving end. Though this process is extremely efficient, the overall performance gains are negligible, as command events represent only a tiny fraction of the total event load Solar must handle.

The vast majority of events routed through Solar come from sources and operators; a simple string cannot represent these arbitrarily complex events. The *data-stream mode*, however, can. The data-stream mode assigns logical id numbers to each member field of an event class. To transmit an event instance, it processes each field in numeric order, serializing the data in the field with a `DataStream`, a low-level Java stream type used to read and write Java class files. If a field represents a container or a relation to another object, it is processed depth-first. On the receiving end, the mode instantiates a new

24

instance of the event class and packs it with the streamed field content. The data-stream mode preserves object identity on reconstruction, and therefore is able to handle data structures containing circular references and fields that point to the same object instance.

Using the data-stream mode, we were able to reduce object serialization times by an average of 68% relative to standard Java serialization. To our surprise, however, the data-stream mode also increased the network transfer size of objects by almost 30%. Our use of shared serialization reduces the importance of serialization speed; in our tests, the rise in the amount of data that must be transmitted for each event completely negated the benefits of faster serialization. Moreover, tests conducted in low-bandwidth environments would suffer an even greater penalty due to the increased network traffic. Thus our attempt to enhance throughput using an alternative data-transfer mechanism failed.

7.1 Optimization Conclusions

While not always successful, our attempts to optimize Solar produced some valuable conclusions. It is now clear that standard Java serialization uses a condensed format that employs advanced buffering to significantly boost performance. To make appreciable throughput gains, then, it is not enough to create a transfer mode that can package and unpackage objects faster than standard serialization; the mode must also employ tune its buffering for the network and output compact data to compete. It may even be necessary to limit the complexity of context data so that intricate serialization can be avoided. For example, we found that simply adding object identity preservation to our data-stream mode decreased its speed in converting objects to and from a network-ready form by over 20%, and increased the network size of its output by 5%. If considerations like these could be ignored, the serialization process could become dramatically more efficient. It might also be worthwhile to investigate lessening the amount of data that must be transferred by tracking changes to successive events in a stream and only transmitting data that differs between them.

8   Interoperability

Interoperability is a key requirement for any ubiquitous-computing infrastructure. In an environment consisting of thousands of independent sensors and devices, it is unreasonable to expect every application to run on a common computing platform or to use a single programming language. Thus, system infrastructure components such as Solar must bear the burden of communicating with heterogeneous clients. Traditionally, interoperability is accomplished in one of two ways:

1. **Standards.** Infrastructure components can use a single platform-agnostic standard for all communication with the outside world. Any application that conforms to the standard can integrate with the system. This approach has been adopted by the emerging *web services* paradigm in cross-platform computing, which uses XML for all inter-machine data transfer. Using a single standard such

as XML simplifies the chore of maintaining cross-platform compatibility, but it lacks flexibility. The chosen standard must be a lowest-common-denominator format to support the greatest number of outside systems, which means that it is likely not ideally suited to the task at hand. Also, relying on a single standard often negates the possibility of optimizing communications when the system happens to be used in a more homogeneous computing environment.

2. **Pluggability.** An alternative to relying on a single standard is to offer applications a choice of several possible communication modes. This approach is more complex than a single-standard mandate, but it offers several advantages: more applications can be supported, applications can choose their preferred data format, and the most optimal format can be used for each client.

Despite the added complexity, we chose to implement the second option in Solar. This option is more in line with the goals we set for the interoperability framework, namely:

- o **Subscriber Choice.** Solar subscribers should be able to choose their preferred method of receiving context events. This contributes to Solar's overall objective of easing the difficult task of context-aware application development as much as possible.

- o **Efficiency.** The interoperability framework should not be a bottleneck in event transmission. It should be possible to build transfer modes with the framework that are at least as fast as standard Java serialization.

- o **Extendibility.** The interoperability framework should be extensible, so that more data formats can be added later, enabling applications to choose the optimal format for their needs. Additionally, creating an extensible framework ensures that the system can evolve over time while maintaining backwards compatibility with older applications.

- o **Complex Context.** Solar sources can publish arbitrarily complex events. The interoperability framework must not impede sources with restrictions on the data they can distribute.

- o **Transparency.** The interoperability framework should be completely transparent to developers. Event types should continue to be represented by standard Java classes, without having to implement special interfaces or logic to be serialized into the various supported data formats. Additionally, event classes that have already been written should not need to undergo modification to work with the new interoperability framework.

- o **Security.** Many events may include non-public fields that contain sensitive information. The interoperability framework should not allow unauthorized access to the data stored in these events.

8.1   Design

How can we access the non-public fields in event objects to serialize them? And even if we are able to access the fields' data, how do we prevent malicious code from also gaining unfettered access to it? Furthermore, how do we accomplish all of this so that it is completely transparent to the event class developer? The answers to all of these questions lie in the power of Java bytecode enhancement.

8.1.1   Bytecode Enhancement

Bytecode refers to the low-level building blocks of compiled Java classes. Just as Java programmers must write source code that complies with the Java language specification for it to compile correctly, so too must Java compilers output bytecode that conforms to the Java bytecode specification for it to run correctly [8].

Bytecode *enhancement* is the process of modifying the bytecode of a class to increase the functionality of that class. Bytecode enhancement is an extremely powerful development technique with some compelling advantages over more traditional methodologies:

- It does not require access to the source code. This flexibility is beneficial in situations in which the source code has been lost or was never provided.

- It is transparent to the developer. Bytecode enhancement takes place after compilation. The developer of the to-be-enhanced classes may not even know it will occur – her write/debug/test cycles are completely unaffected, and the functionality provided by enhancement is given without her having to implement any special interfaces or write special code. In a very real sense, the functionality is provided "for free".

- It is "clean". Parsing and manipulating bytecode is often much more straightforward than attempting to automatically generate or modify source code, though this is a subjective observation rather than an objective fact. Furthermore, careful bytecode modification can preserve the debugging line numbers of the original code, while source-code modification cannot.

- Bytecode modifications can be made through custom class loaders at runtime, removing the need for even an extra after compilation. Our Solar interoperability mechanism currently does not use this capability, but it may in the future.

There disadvantages to dealing directly with bytecode as well:

- Errors are difficult to debug. If an enhancer produces invalid bytecode, the resulting class will simply fail to load or fail to execute properly, often with cryptic or absent error messages.

o   Programming is low-level. Working with bytecode is akin to working in a simple assembly language.

o   Finally, some developers seem to have an aversion to bytecode modification. Their fear, though often irrational, sometimes hinders the adoption of systems that employ bytecode enhancement.

Bytecode enhancement is certainly not suitable for every project. In situations that require absolute transparency, or situations that would normally put too much of a burden on the class developer, though, it can often succeed where all else fails.

8.1.2 Event Modifications

We enhance the bytecode of Solar events to add three new aspects of functionality to each event class: factory operations, metadata operations, and state operations. Transfer modes use these operations to implement custom event serialization and deserialization. All operations are designed for maximum efficiency.

Factory operations are methods enabling an event instance to act as a factory for other instances of the same type. The bytecode enhancer adds a `solarNewInstance()` method to each event class; this method constructs and returns a new instance of the event. The enhancer also adds code to each event class so that when the class is loaded into the Java Virtual Machine (JVM), it registers an instance of itself with the *transfer helper*, a singleton helper object. The transfer helper hashes each registered instance based on the instance's type. When an event is sent over the network, the code on the receiving end of the transfer uses the transfer helper to create a new instance of the proper event class; this instance can then be packed with the streamed data. To create the requested instance, the transfer helper simply looks up its hashed event object for the given class and uses the object's `solarNewInstance()` method to construct a new event. Using the transfer helper and the `solarNewInstance()` method, we avoid Java reflection, a relatively slow means of dynamically inspecting and creating an object given its class. Tricks like this one are partially responsible for the superior speed of the data-stream transfer mode built on the interoperability framework; the data-stream mode is discussed in Section 7.

Figure 13 displays a source code representation of the factory operations added to each enhanced class. The enhancer implements the factory operations in bytecode; therefore, the given source is only an approximation of the actual operations.

```
public class TemperatureEvent
   implements Transferrable         // interface added by enhancer
{
   static
   {
      TransferHelper.getInstance().register(new TemperatureEvent());
   }

   public Transferrable solarNewInstance()
   {
      return new TemperatureEvent();
   }

   // remainder of class definition omitted…
}
```

Figure 13: Source-code view of the bytecode added during class enhancement to implement factory operations.

The metadata operations added to enhanced events are also designed to circumvent Java reflection. These operations enable transfer modes to gather information about the member fields of an event class so that it can choose the most efficient network format for the data in each field. Figure 14 depicts an approximation of the source code for some of the metadata operations on hypothetical `TemperatureEvent` and `UnitTemperatureEvent` classes.

Metadata operations use *absolute ids* rather than names to refer to class member fields. During enhancement, we assign each member field of an event class a *relative id*, where relative ids are monotonically increasing integers starting at 0. At runtime, we calculate the *absolute id* of a field by adding its relative id to the total number of fields it inherits from other classes further up the inheritance hierarchy. We do not calculate absolute ids at enhancement time so that modification to the field count of a superclass does not require re-enhancement of it subclasses. The example displayed in Figure 14 may clarify the use of absolute and relative ids.

Ids are more useful than field names in several respects. For example, they can be used in fast `switch` statements, and multiple fields can be specified at once as a range from a low id number to a higher id number. The state operations discussed below take advantage of this latter capability of ids to implement efficient state transfer between the event and the transfer mode.

Given the absolute id of a field, the metadata operations of an event can return the field name, the field class, and a symbolic constant indicating the general field type, such as `INT`, `MAP`, or `ARRAY`. This constant is often more useful than knowing the exact field class, as it can be used in `switch` statements. Metadata is used by most transfer modes during both event serialization and deserialization.

```
public class TemperatureEvent
   implements Transferrable
{
   private double temp;
   private long timestamp;

   public int solarGetFieldCount()
   {
      return 2;
   }

   public String solarGetFieldName(int field)
   {
      switch(field)
      {
      case 0: return "temp";
      case 1: return "timestamp";
      default: throw new IllegalArgumentException();
      }
   }

   public int solarGetFieldTypeCode(int field)
   {
      switch(field)
      {
      case 0: return Transferrable.DOUBLE;
      case 1: return Transferrable.LONG;
      default: throw new IllegalArgumentException();
      }
   }

   // remainder of metadata methods and class definition omitted…
}

public class UnitTemperatureEvent
   extends TemperatureEvent
{
   private String units;

   public int solarGetFieldCount()
   {
      return super.solarGetFieldCount() + 1;
   }

   public String solarGetFieldName(int field)
   {
      int relId = field − super.solarGetFieldCount();
      if(relId < 0)
         return super.solarGetFieldName(field);

      switch(relId)
      {
      case 0: return "units";
      default: throw new IllegalArgumentException();
      }
   }

   // remainder of metadata methods and class definition omitted…
}
```

Figure 14: Source-code view of some of the bytecode added during class enhancement to implement metadata operations.

The final category of functionality added during event enhancement deals with state transfer. We add methods to the event class so that the data in each field can be read or written by the event-transfer system. Like metadata methods, these methods use absolute field numbers; unlike metadata methods, however, they do not simply return the requested data. Rather, they use a system of callbacks on an *output manager* or an *input manager*.

An output manager is used to retrieve state from an event instance during serialization. The transfer mode passes the output manager and a range of field ids to the added `solarProvideFields()` method of the event being serialized. This method first checks with the Java security manager to make sure system security policy grants the calling code permission to access internal event state. Next, for each absolute field id in the given range, it provides the value of the corresponding event field to the output manager. The output manager typically serializes the value and outputs it to the network. This process is diagrammed in Figure 15.
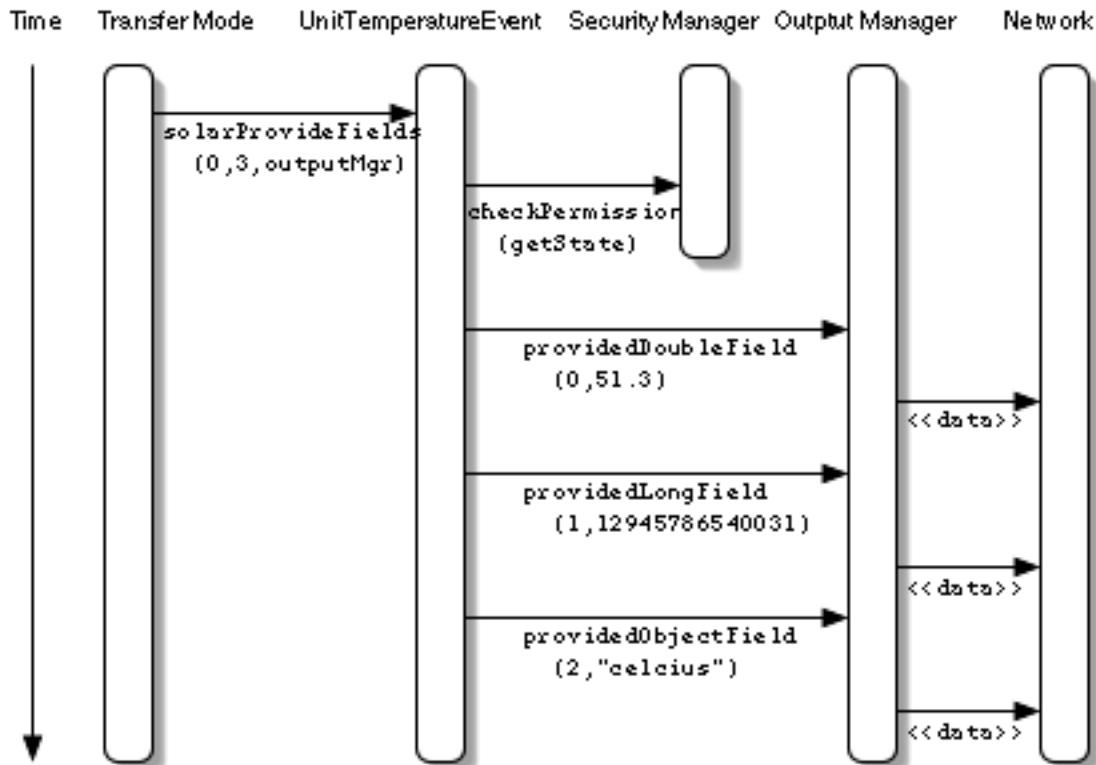

Figure 15: Possible method call sequence for outputting the state of a `UnitTemperatureEvent`.

The input manager is the complement of the output manager; it is used to load state into an event instance during deserialization. The transfer mode performing the deserialization constructs an event instance using the transfer helper discussed earlier. The mode then calls the added `solarReplaceFields()` method of the newly constructed event, passing it the input manager and a range of field ids. The `solarReplaceFields()` method uses the Java security manager to ensure that the

calling code has permission to set internal event state. For each field id in the given range, it then calls the one of the input manager's `replacing()` methods with the field id as an argument. The `replacing()` method returns the value the field should be set to; the value is typically pulled from the serialized data stream being read from the network. Figure 16 illustrates this system in action.
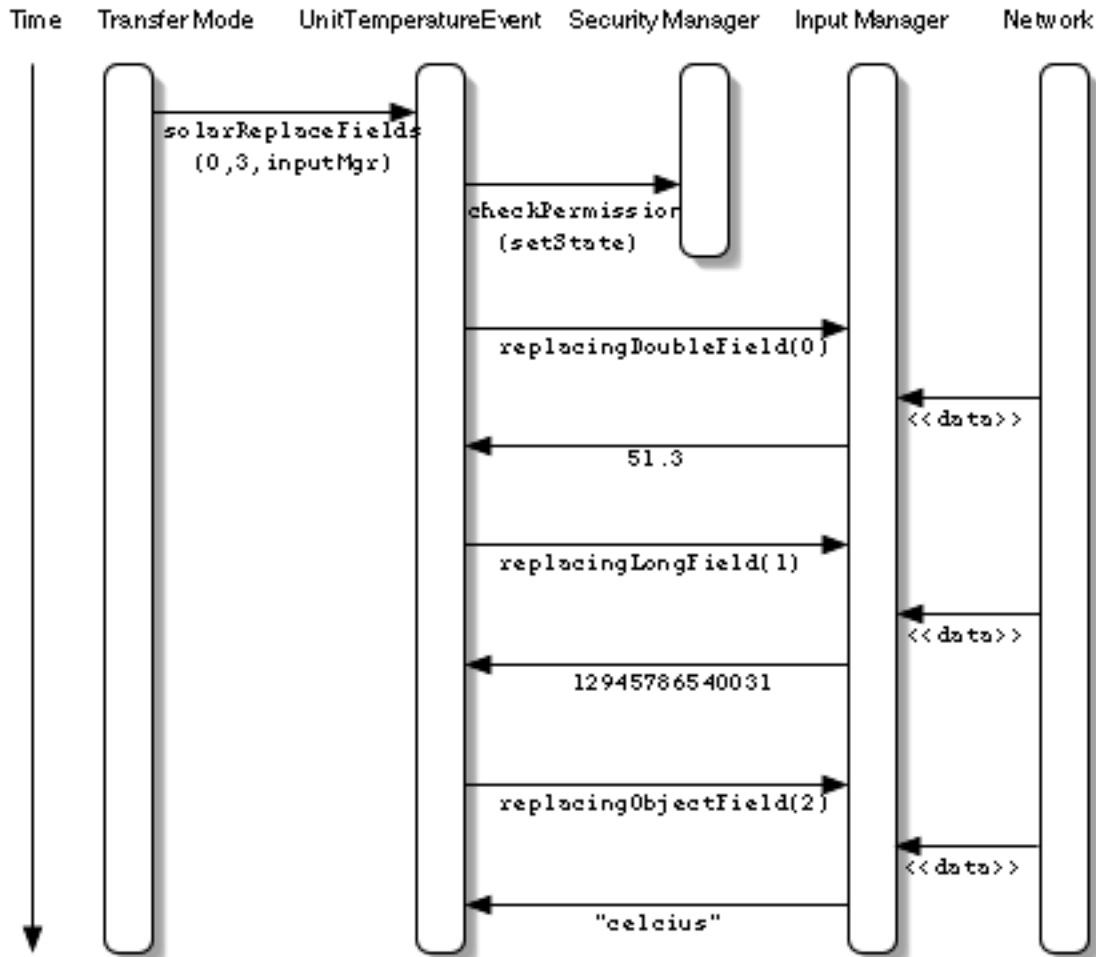

Figure 16: Possible method call sequence for inputting the state of a
`UnitTemperatureEvent`.

8.1.3 Security

The purpose of implementing all state access through a series of callbacks is security – or, more accurately, security combined with efficiency.

The interoperability framework integrates with Java's standard security APIs to ensure that malicious code cannot access the information stored in non-public event fields. It defines special permissions that the system's security policy must grant to any code-base that will set or retrieve data from an event instance through the methods added by the bytecode enhancer. Checking these permissions every time a transfer mode requires access to an event field, however, would be prohibitively slow; one of the primary goals

of the framework is to be as efficient as possible. Thus, we architected the system to allow groups of fields to be accessed with a single method call, and therefore a single permission check.

Java supports neither pass-by-reference semantics nor methods with variable-length parameter lists. Therefore, there are only two options for methods that wish to return multiple values or take arbitrary parameters: pack the information into a single data structure, or implement a series of callbacks. The data-structure approach is complex and relatively inefficient, primarily due to the primitive/object duality that exists in Java. Standard Java data structures, such as Collections, Sets, and Maps, can store objects but not primitives. Primitives must therefore be wrapped in simple value objects to be stored; the process of wrapping a primitive in an object is often called "boxing" and is inefficient due to the required allocation of heap space during object construction, and the subsequent need to garbage collect the objects when they are no longer referenced. Thus, to maintain efficiency in our interoperability framework, we were left with the callback approach illustrated above. Figure 17 gives a source-code view of the state operations, including the security checks and the callback system.

```
public class UnitTemperatureEvent
   extends TemperatureEvent
{
   public void solarProvideFields(int start, int finish,OutputManager outputMgr)
   {
      System.getSecurityManager().checkPermission(TransferPermission.GET_STATE);
      for(int i = start; i < finish; i++)
         solarProvideField(i, outputMgr);
   }

   protected void solarProvideField(int field, OutputManager outputMgr)
   {
      int relId = field — super.solarGetFieldCount();
      if(relId < 0)
         return super.solarProvideField(field, outputMgr);

      switch(relId)
      {
      case 0: outputMgr.providedObjectField(field, units);
      default: throw new IllegalArgumentException();
      }
   }

   public void solarReplaceFields(int start, int finish, InputManager inputMgr)
   {
      System.getSecurityManager().checkPermission(TransferPermission.SET_STATE);
      for(int i = start; i < finish; i++)
         solarReplaceField(i, inputMgr);
   }


   continued on the next page…
```

33

```
   protected void solarReplaceField(int field, InputManager inputMgr)
   {
     int relId = field — super.solarGetFieldCount();
     if(relId < 0)
        return super.solarReplaceField(field, inputMgr);

     switch(relId)
     {
     case 0: units = inputMgr.replacingObjectField(field);
     default: throw new IllegalArgumentException();
     }
   }

   // remainder of class definition omitted…
}
```

Figure 17: Source-code view of the bytecode added during class enhancement to implement state operations.

8.2 Transfer Modes

Section 7 presented the command mode and the data-stream mode, two transfer modes built on the interoperability framework and designed for performance. We also created one other transfer mode – the *XML mode*. This transfer mode is not built to improve performance, but to improve interoperability. As its name implies, it transmits events as XML documents, enabling any computing platform with a compliant XML parser to take advantage of Solar's context-dissemination capabilities.

9 Related Work

The callback system used in the interoperability framework is modeled after portions of the Java Data Objects specification from Sun Microsystems [10].

JECho is a Java-based communication infrastructure for collaborative high performance applications [2]. JECho includes some of the same data-transfer optimizations we implemented for Solar in the course of this thesis, as well as many others that we should consider for future work. Our novel approach of using bytecode modifications to access internal object state allows us to maintain higher development transparency for our serialization system than that of JECho, and should allow us to eventually surpass JECho in performance as well.

The Sienna project conducted a detailed performance analysis of matching algorithms for content-based routing [1]. There is little information available on the analysis of overall system performance in Sienna, however, or in other highly distributed information dissemination systems.

Similarly, few testing frameworks are designed to run performance tests on highly distributed infrastructure systems. The vast majority of test harnesses either presuppose a specific type of application, such as J2EE [5] or CGI [4], or, as in the case of the popular JUnitPerf library [7], do not consider tests spanning multiple hosts.

10   Conclusions

Though there is still a lot of work to be done, our effort to increase the performance and interoperability of Solar was, for the most part, successful. We improved many aspects of Solar's performance and gave it the ability to communicate with heterogeneous systems. More importantly, we gained insight into what facets of Solar must be optimized, and laid the architectural foundations for future optimizing software extensions.

In addition to accomplishing our stated goals, our work produced two interesting side effects: the performance testing framework and the interoperability framework. The performance testing framework appears to surpass other testing software in its ability to handle unknown, highly distributed, highly configurable applications. Its generic, modular architecture should also allow it to expand to meet the challenges of testing future systems. Frameworks like this one may be increasingly important as the role of the network expands in computing.

The interoperability framework is interesting for two reasons. First, it teaches us about the power and utility of bytecode enhancement. We created a fairly complex infrastructure that enables the serialization of instances of certain classes to almost any format, and yet to the authors of these classes the process is completely transparent. Moreover, exploiting bytecode modification allowed us to avoid other less efficient forms of state access, resulting in a data-stream serialization mode that was three times faster than standard Java serialization at converting objects to and from a network-ready format.

Second, the interoperability framework itself could benefit many applications that must communicate with heterogeneous systems. The framework is only very loosely coupled to Solar, and could easily be adapted for other uses.

11   Future Work

Solar is poised to become a viable option for context dissemination in ubiquitous computing environments. It offers extreme flexibility and simplifies the development of context-aware application tremendously. We must, however, continue to improve Solar's performance and further expand its interoperability. Possible measures towards these ends include:

- **Additional Test Cases.** More test cases are needed to evaluate Solar's performance under varying conditions. Comparing the results of tests run under different network speeds would be particularly interesting, as would analyzing results from complex tests designed to model real-world usage scenarios.

- **Optimizations.** The optimizations we have implemented thus far only scratch the surface. Solar is still in prototype phase, and opportunities for further optimization abound.

- **Further Profiling.** As Solar performance improves and fine-tuning becomes increasingly important, profiler data will be an invaluable tool in pinpointing bottlenecks and identifying further optimization prospects.

- **Additional Transfer Modes.** The library of transfer modes should be expanded, giving applications a wider variety of data formats to choose from.

- **Extending Interoperability.** The interoperability framework we developed allows non-Java applications to take advantage of Solar as a context delivery mechanism; they are prevented from deploying their own non-Java operators, however. Options to expand interoperability to Solar operators should be investigated. For example, XSLT or XQuery, used in conjunction with the XML transfer mode we created, may be able to act as simple, platform-neutral operator languages, though this approach has drawbacks [12].

The performance testing framework also has the potential to become a useful tool in its own right. The following improvements will help it to excel as a general harness for testing highly distributed, highly configurable systems:

- **Error Reporting.** The current implementation does not capture enough information about errors that occur in the processes that it spawns.

- **Statistics.** The testing framework calculates the min, max, mean, and standard deviation of numeric results. Its abilities in this area should be expanded to encompass other statistics and calculations.

- **Network Usage.** It would be useful to track the amount of data sent through the network during a test.

12   References

[1]   Antonio Carzaniga, David Rosenblum, and Alexander Wolf. Design and Evaluation of a Wide-Area Event Notification Service, Volume 19. ACM Transactions on Computer Systems, 19(3):332-383, August, 2001.

[2]   Yuan Chen, Greg Eisenhower, Karsten Schwann, and Dong Zhou. JECho – Interactive High Performance Computing with Java Event Channels. International Parallel and Distributed Processing Symposium, 2001.

[3]   Guanling Chen and David Kotz. Supporting Adaptive Ubiquitous Applications with the Solar System. Technical Report TR2001-397, Dept. of Computer Science, Dartmouth College, May, 2001.

[4]   Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/interface.html

[5]   Java 2, Enterprise Edition. http://java.sun/com/j2ee/

[6]   JProbe is a product of Sitraka. http://www.sitraka.com/software/jprobe

[7]   JUnitPerf. http://www.clarkware.com/software/JUnitPerf.html

[8]   Tim Lindholm and Frank Yullin. The Java Virtual Machine Specification, Second Edition. Addison Wesley, May 1999.

[9]   Arun Mathias. SmartReminder: A Case Study on Context-Sensitive Applications. Technical Report TR2001-392, Dept. of Computer Science, Dartmouth College, June, 2001. Senior Honors Thesis.

[10]   Craig Russell. Java Data Objects Specification, version 1.0. Sun Microsystems, April 2002. http://access1.sun.com/jdo

[11]   Mark Weiser. The Computer for the 21st Century, volume 265. *Scientific American*, 265(3):66-75, January, 1991.

[12]   Abram White. XSLT and XQuery as Operator Languages. Technical Report TR2002-429, Dept. of Computer Science, Dartmouth College, June 2002.