OverView: Generic Visualization of Large-Scale Distributed Systems

Jason LaPorte Travis Desell Carlos Varela

overview@cs.rpi.edu
http://wcl.cs.rpi.edu/overview/

August 30, 2006

Abstract

Visualization aids developers in conceptual understanding, testing, and debugging of distributed systems. Many of these systems, due to their complexity, are programmed using high-level abstractions (such as actors, ambients, processes, or virtual machines); and while tools are lacking for such visualization in general, there is a specific need for visualization of very large distributed systems, including thousands or tens of thousands of these *entities*. OverView is an entity-specification driven software toolkit which allows visualization of these systems; preserving the high-level concepts within and displaying their important properties, such as location, remote communication, and migration, both online (that is, in real-time) and offline. OverView's architecture is generic, in that different abstractions may reuse the same visualization modules, requiring only a change in the entity-specifications that map high-level code to the visualization abstractions.

1 Problem Description

Distributed systems (for example, a computer program that is distributed across a network) have recently become very popular, due to their versatility and the relatively low expense of increased computation power through networking. However, these systems are extremely complex, and thus the burden on programmers and researchers to develop these systems has increased.

For example, one might wish to see the activity, computationally speaking, of the computers comprising SETI@home[1].[1] Perhaps one wants to view the interactions between the computers running a high-performance simulation across several computational clusters. You might also want to see, visually, a BitTorrent swarm as its activity spikes and dwindles over time. More intriguing is the idea of a distributed program running on mobile phones, the runtime of which may migrate from the phone to nearby servers and back again as the owner of the phone walks around a city, or if the battery becomes critically low.

Developers and maintainers of such systems can gain a productivity boost if they can have a means of visually analyzing these systems; furthermore, such a tool can be used as an aid to teach students concepts of distributed systems. Therefore, the creation of a tool to visualize distributed systems is a useful and worthwhile endeavor to undertake.

1.1 Specific Problems in Distributed Systems Visualization

It is worth describing some of the specific problems inherent in visualizing distributed systems, and they

[1] At a given time, there are one hundred thousand personal computers working on a computation. http://setiathome.berkeley.edu/

are outlined in this section.

1.1.1 Scalability

Scalability is a paramount problem, and of particular interest to us, as we wish to visualize tens of thousands of entities across potentially thousands of computer systems. Addressing scalability covers two related problems; firstly, that of the scalability of the architecture; and secondly, that of the scalability of the visualization. The latter will be included in Section 1.1.4.

Architecture scalability is the ability for the software framework itself to scale. The difficulty is that of the client bottleneck: if a distributed system is running on thousands of physical machines, then the visualization tool must receive events, directly or indirectly, from each machine. This is extremely expensive in terms of network bandwidth and client processing time; and unless steps are taken, the visualization may be needlessly inefficient, or else be too slow to be practical or possible.

1.1.2 Genericity

Genericity refers to how well a single type of visualization can be applied to a heterogeneous set of problems. For example, we may wish to visualize mobile object-based systems, actor-based systems, mobile ambient-based systems, mobile virtual machine-based systems, or other abstract concepts of distributed systems, all using the same framework.

Furthermore, it also refers to how well different languages may be visualized with the system. For example, visualization of Java-based systems, as well as C and MPICH-based systems.

1.1.3 Online and Offline Visualization

Most visualization tools offer offline visualization; that is, constructing a visualization from a log or description file. While this is a worthwhile design goal, distributed systems are far too dynamic for such a system to be very useful. Therefore, it is important to have a visualization tool that runs online—that is, in real-time, while the distributed system is running—so that it is more useful for those who use it; this carries the added necessity of runtime profiling that is not expensive or intrusive to the system. It is important to retain the ability to use the visualization offline, so that one can record and replay a visualization if required.

1.1.4 Visualization Clarity

A visualization's purpose is to enhance one's ability to understand a system, by making statistical information visual so that our brains, which are already built and fine-tuned to process images, may better comprehend the information at hand. Thus, it is also of vital importance that the visualization is clear and easy to understand at a glance. This means that the visualization must be intuitive, and that it must scale well: that is, it must be able to make small systems (a few dozen to a few hundred elements) just as understandable as large systems (thousands to hundreds of thousands of elements).

Unfortunately, this not a simple problem, since small-scale visualizations tend to be quantitative in nature (as detail is important), while large-scale visualizations tend to be qualitative (as trends become more important than details). These two goals are generally mutually exclusive, and so means to change the type of visualization must be put into place.

2 Approach

OverView[2, 3][2] is a toolkit which permits visualization of distributed systems written in Java, in order to better enable those interested in these systems to understand them. The *OverView Instrumenter* (OVI) allows the abstraction of Java method invocations (without modifying the code of the existing Java program) into a small set of events, which are sent over a network to the *OverView Presenter* (OVP), which interprets the events and displays a meaningful, interactive, graphical representation of the state of the distributed system. This is shown in Figure 1.

[2] At the time of writing, the current version of OverView is 0.3.3.

In OverView, Java method invocations are mapped to events, which represent a high-level abstraction of a program's execution. The mapping is defined in an *Entity Specification Language* (ESL) file, which is read by OVI. OVI then instruments (that is, inserts new instructions into) the compiled Java bytecode of the program, which causes it to send those events over a socket to any listening OVP when the program is run. This instrumentation does not significantly effect the runtime of the program, it simply adds asynchronous event-transmitting behavior.

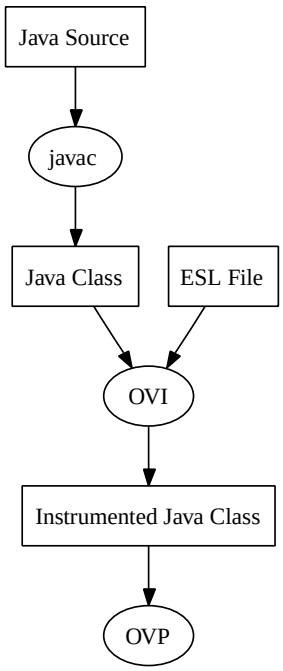

Figure 1: A high-level description of how OverView is used.

2.1 The Entity Specification Language

The Entity Specification Language is a simple declarative language, which is used to map Java method invocations to OverView events, so that one can abstract any Java program into a set of cues that OverView can use to render a visualization. The ESL grammar is provided in Figure 2, and a sample ESL file, which handles creation and communication events for SALSA actors, can be found in Figure 3. The benefit of using such a language is twofold: firstly, it allows us to make our visualization generic to any Java program, regardless of how it is implemented. Secondly, if one desires to visualize different parts of the same program, or the same program at different granularities (for example, at the object-level or at the level of some higher-order composition of objects), one may do so by simply instrumenting the program using different ESL files.

OverView currently supports the following types of events:

- **Creation**, which represents the creation of an entity.
- **Migration**, which represents the movement of an entity from one entity or container to another.
- **Deletion**, which represents the elimination (either manually or via garbage-collected) of an entity.
- **Communication**, which represents when one entity sends information to another (which can represent a method call, a remote method invocation, an message-sending operation, or so on).
- **Split**, which is when one entity begets another (e.g. for load balancing)[4].
- **Merge**, the opposite of a split operation, which merges one entity into another.
- **Error**, which displays an error message in relation to some entity.

2.2 The OverView Instrumenter

OVI itself has two parts: an instrumenter, and an *Integrated Profiling Agent* (IPA). The instrumenter itself uses ESL entity specifications as a road map to insert event-sending bytecode into a compiled Java program. The code that is inserted is the IPA: what it does is simply send a specified event to a listening

Entity	::==	**entity** IDENTIFIER **is** *Name EntityBody*
EntityBody	::==	**{** *UniqueByDeclaration* (*WhenDeclaration*)* **}**
UniqueByDeclaration	::==	**unique by** *Value*;
WhenDeclaration	::==	**when** (**start** | **finish**) *MethodSpecification* -> *EventDeclaration* [*ExceptionSpecification* (**,** *ExceptionSpecification*)*];
MethodSpecification	::==	IDENTIFIER ([*Parameter* (**,** *Parameter*)*])
ExceptionSpecification	::==	**on exception** IDENTIFIER -> *EventDeclaration*
Parameter	::==	*Type* IDENTIFIER
Type	::==	IDENTIFIER (**.** IDENTIFIER)*
EventDeclaration	::==	*EventType* ([*Value* (**,** *Value*)*])
EventType	::==	**Creation** | **Migration** | **Deletion** | **Communication** | **Split** | **Merge** | **Error**
Value	::==	LITERAL | **exception** | (**entity** | IDENTIFIER) (**.** *ValuePart*)+
ValuePart	::==	IDENTIFIER [([*Value* (**,** *Value*)*])]

Figure 2: The ESL grammar.

```
entity UniversalActorState is salsa.language.UniversalActor$State {
        when start addClassName(java.lang.String className)
                -> creation(this.getUAL().getIdentifier(),
                            this.getUAL().getHostAndPort());

        when start putMessageInMailbox(salsa.language.Message arg0)
                -> communication(arg0.getSourceName(),
                                 arg0.getTargetName());
}
```

Figure 3: An example ESL file. This particular file is for generating events from SALSA actors.

OVP when it is triggered (during a method invocation, for example).

The instrumenter uses Apache's *Byte-Code Engineering Library* (BCEL)[3], which facilitates the insertion of new bytecode into existing programs. The profiling agent is of tremendous importance in the topic of scalability, and will be further discussed in Section 5.

2.3 The OverView Presenter

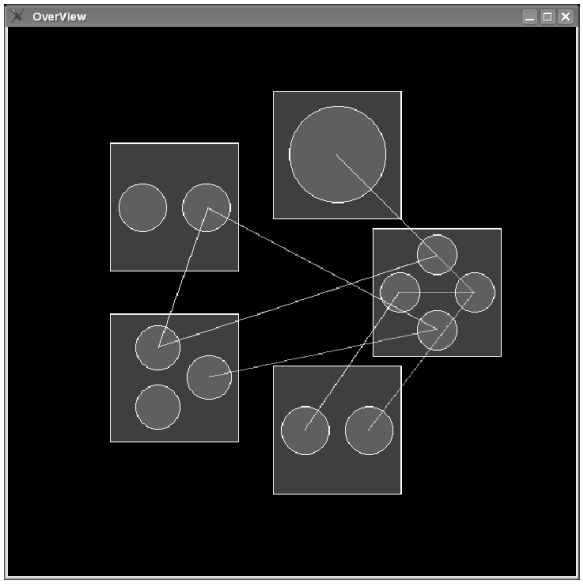

Figure 4: A screenshot of OVP, with 5 containers and 12 entities.

OVP, when invoked, will create a number of client connections to IPAs and listen for incoming events, which are multiplexed and translated into a visualization. Most of the software consists of the visualization itself, and the various means of controlling it. Its design is shown in Figure 6.

The visualization is currently a simple and intuitive one, though it does not scale particularly well: it remains intuitive enough for dozens of elements (Figure 4), but becomes excessively cluttered and difficult

[3]http://jakarta.apache.org/bcel/index.html

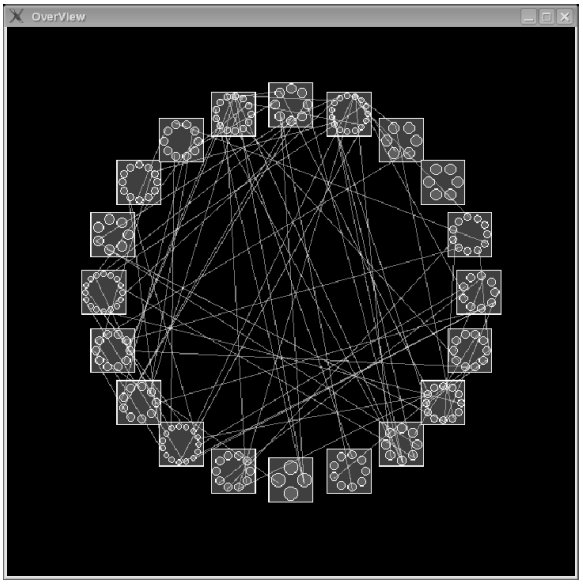

Figure 5: Another screenshot of OVP, with 20 containers and 200 entities.

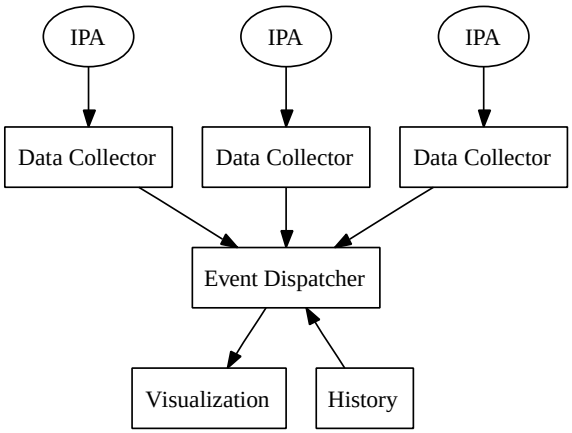

Figure 6: The structure of OVP. Events, which are sent by multiple profiling agents, are collected by Data Collectors, one for each agent. These are sent to a single Event Dispatcher, which collects the events and forwards them to a Visualization module. A History module keeps a pointer which is used to tell the Event Dispatcher which events to forward.

to understand with hundreds of elements (Figure 5), far short of our goal of thousands or tens of thousands. Two basic concepts exist within our visualization: *entities* and *containers*. Entities are merely some self-contained unit of computation, such as an object, an actor, a process in a physical computation environment. Containers are just that: they contain entities. For example, a computer on which processes run, or a virtual machine in which actors perform. Entities can exist within containers, within other entities (such as an actor on a mobile virtual machine), or can simply float around, being without containment of any kind. When two entities communicate, a line is drawn between them. The line will decay over time, so the brighter the line, the more frequently the two entities communicate. Whenever an entity moves, or splits, or merges, its motion is interpolated so that the visual cues may provide the viewer with a more intuitive understanding of what is happening. The user may use the mouse to zoom in or out of the visualization, pan it around, or fast-forward, pause, or rewind through time, like you might do with a VCR.

3 Preliminary Results

OverView has been tested on a number of programs, such as visualizing recursive computations. It has also been shown to visualize *SALSA*[5][4] (Simple Actor Language System and Architecture) programs, such as heat distribution through a bar[5], across several computer systems. However, it remains untested on "real-world" applications (such as those described in Section 1) and on very large networks. Rensselaer has put much effort into creating large-scale computing infrastructure, however; and with some interesting programs being written to make use of it (for example, an astronomy simulation program), we intend to conduct such tests soon.

SALSA in particular benefits greatly from OverView, as SALSA itself can be instrumented, removing the need to instrument SALSA source files. This makes visualization of SALSA programs with OverView as simple as using a particular SALSA version to compile your source code.

One of the specific choices made in the current version of OverView is it's distinctive ring structure. Containers are placed in a large ring around the window (which is, of course, able to be panned and zoomed), while entities are placed around rings within their parents. We had looked at several different placement mechanisms, such as random and book-style (left-to-right and top-to-bottom), and this one has demonstrated itself to be better than all the rest for the following reasons: firstly, it minimizes ambiguous communications: that is, communications overlapping multiple entities, so you cannot tell at a glance which it refers to; this is a problem that appears often with a grid-style setup since so many communications are horizontal or vertical. Secondly, it can display a hierarchy of elements intuitively even with a the naive placement of new elements at the end of the list, allowing it to support recursive computations as well as any other kind (Figure 7). It's downsides include a lot of wasted space (in the middle of the ring), and it doesn't scale up, because if you're looking at one side of the ring in detail, you can't be looking at the other. However, for small networks, it works very well.

Another implementation difficulty was the History module. It was implemented in the following way: when events are collected from IPAs, they are buffered into a list of events. A pointer is kept that points somewhere in the list; by default, it is always at the end, but it can be moved linearly as well. The difficulty in implementation was that, since we require the visualization to remain consistent, we cannot allow random-access in the visualization (via, for example, checkpointing) since this would cause a "jump" in the visualization. So, instead, we must have methods for doing events, and for undoing them, to restore the state that occurred before the event was received. Aggregations of these actions allow us to move to any temporal point in the visualization. Some events (such as communication) are instantaneous, and have no lasting effect on the visualization, so they are reversible (that is, doing is the same as

[4]SALSA is an extension to Java that allows programming using the Actor model. http://wcl.cs.rpi.edu/salsa/

[5]You can see a video of this online, at http://wcl.cs.rpi.edu/overview/media/heat.avi

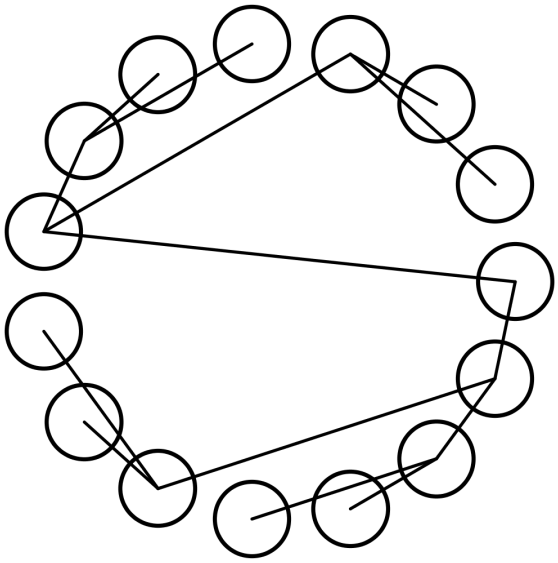

Figure 7: An example of the ring visualization on a binary tree, placing new elements (in a preorder traversal) at the end of the list.

undoing). Others, however, such as deletion, remove the context in which the entity existed (for example, we must know where the entity was located before it's deletion to be able to undo it), so we had to introduce a mechanism to store this context. We store the context when the event is first played; it should be noted that this is possible since an event must *always* be played forwards before it can possibly be played backwards. This is because the tape is linear and we always start at the beginning.

4 Prior and Related Work

4.1 Prior Work on OverView

At an earlier stage in OverView's development, it was implemented as a plug-in for the Eclipse[6] IDE. This was motivated by the desire of spring-boarding off of Eclipse's architecture, and with the desire of possibly using OverView as a debugging component to SALSA programs, within Eclipse's debugging software. However, the author saw this as a goal too far in the future, and separated OverView from Eclipse to make it smaller, simpler, more portable, and to open up new venues of development (for example, visualization from within a web browser via a Java applet).

[6]http://www.eclipse.org/

4.2 Related Work in Visualization

Frishman and Tal's *Visualization of Mobile Object Environments*[6] demonstrates a software project similar to our own, but very different in mechanism. While it is limited in genericity (as it relies upon a particular implementation of Mobile Objects in Java, for execution and profiling purposes), it scales much better than OverView does, at present. It suggests providing the user a means to select a "focus," that is, select which parts of the visualization are of interest. Then, the software visualization may "filter out" unrelated or unimportant parts of the visualization. If this filter is moved close to the data source (that is, into the profiling agent), an immediate savings on network bandwidth and client framerate (that is, the time it takes OVP to construct the visualization per frame) will result. It should be noted that such an approach affects not only the architecture, but also the visualization itself.

Benjamin Fry's master thesis, *Organic Information Design*[7], demonstrates that while static visualization systems are well understood (as they've been developed over several hundred years), dynamic visualization systems are still in their infancy. He argues that by taking cues from biological systems (such as metabolism, growth, homeostasis, and reproduction), one can design a more dynamic and scalable visualization that retains intuitiveness. While these designs tend to provide an excellent qualitative representation of a system, which is desirable in large or very dynamic systems, they falter in quantitative measurements, which may be desirable at smaller scales.

5 Future Work and Conclusions

Scalability has not yet been addressed in the software to any significant degree, either in architecture or in visualization. To address architecture scalability concerns, we are experimenting with focus-based approaches described in Section 4.2, along with an event filtering and publish-subscribe mechanisms in each IPA. We are also considering the possibility of using a hierarchical event distribution model to reduce network bandwidth. Events, which are currently being sent as serialized Java arrays of strings, may be changed to some other format for efficiency and for genericity (for languages other than Java).

To address visualization scalability concerns (which are, at the moment, possibly the most limiting factor of the software), we are experimenting with various prototypes of alternate means of visualizations that take cues from the lessons learned from Organic Information Design (Section 4.2) and our own prior experimentation. Examples of possible faults in the current visualization are:

- Nested layers of entities being drawn inside of each other, while intuitive, does not scale well at all, particularly when considering communication between them. Alternate representations of containment are being examined, such as placement and coloration.

- Communicating events draw lines between them, and current mechanisms do well to demonstrate frequency of communication, but not bandwidth of communication. We are interested in mechanisms that, for example, alter the line thickness based on how much information is sent. It should be noted that this is nontrivial, as there must be an ESL abstraction for how much data is being sent, before such a thing may be implemented.

- The visualization may benefit from being more interactive. For example, being able to reposition entities using the mouse, and saving the created layout for future visualizations.

In terms of on- and offline visualization, offline visualization has not yet been implemented. Additionally, dynamic paging of visualization data is a necessity, as leaving a visualization running for too long will crash Java by using up all available heap space. A mechanism must exist whereby the log file is placed in secondary storage, and only the currently used portion is in main memory.

We are constantly evaluating which events OverView supports, so as to have the fewest events necessary; this is to simplify the program and to better enable reasoning about it.

The difficulties inherent in designing a good visualization tool, the potential usefulness of such a system, and the apparent lack of other software of the kind imply that there is a need for distributed visualization tools. While our current results show us that our tool, OverView, does not address the problem of scalability to the degree we feel it should, it takes a unique approach to the problem of genericity: by using the Event Specification Language in conjunction with the OverView Instrumenter, we can profile any Java application unintrusively, freeing the user to visualize any type of system, using any abstraction, at any desired granularity. Online and offline visualization are both important, and our approach of runtime visualization through the OverView Instrumenter's Integrated Profiling Agents (which gather data about the system's state and transmit it) allow us to easily manage both. Finally, the problem of having a clear, intuitive, and understandable visualization is addressed by the OverView Presenter, which displays a graphical representation of the state of the distributed system. We are excited about the promise that OverView has as a useful, generic visualization tool, and hope that as it is further developed, it will prove to be of use to others as much as it is useful to us.

References

[1] W. T. Sullivan, III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project SERENDIP data and 100,000 personal comput-

ers. In C. Batalli Cosmovici, S. Bowyer, and D. Werthimer, editors, *IAU Colloq. 161: Astronomical and Biochemical Origins and the Search for Life in the Universe*, pages 729–+, January 1997.

[2] Harihar N. Iyer, Abe Stephens, Travis Desell, and Carlos Varela. OverView — dynamic visualization of java-based highly reconfigurable distributed systems. Technical report, Rensselaer Polytechnic Institute Worldwide Computing Laboratory, August 2003.

[3] T. Desell, H. Iyer, A. Stephens, and C. Varela. OverView: A framework for generic online visualization of distributed systems. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS 2004), eclipse Technology eXchange (eTX) Workshop*, Barcelona, Spain, March 2004.

[4] Travis Desell, Kaoutar El Maghraoui, and Carlos Varela. Malleable Components for Scalable High Performance Computing . In *Proceedings of the HPDC'15 Workshop on HPC Grid programming Environments and Components (HPC-GECO/CompFrame)*, pages 37–44, Paris, France, June 2006. IEEE Computer Society.

[5] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001.

[6] Yaniv Frishman and Ayellet Tal. Visualization of mobile object environments. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 145–154, New York, NY, USA, 2005. ACM Press.

[7] Benjamin Fry. Organic information design. Master's thesis, Massachusetts Institute of Technology, May 2000.