Organic and Hierarchical Concentric Layouts for Distributed System Visualization

Jason LaPorte and Carlos Varela

Abstract— Distributed systems, due to their inherent complexity and nondeterministic nature, are programmed using high-level abstractions, such as processes, actors, ambients, agents, or services. There is a need to provide tools which allow developers to better understand, test, and debug distributed systems. OverView is a software toolkit which allows online and offline visualization of distributed systems through the concepts of *entities* and *containers*, which preserve the abstractions used at the programming level and display important dynamic properties, such as *temporal* (that is, when entities are created and deleted), *spatial* (that is, entity location and migration events) and *relational* (that is, entity containment or communication patterns).
In this paper, we introduce two general layout mechanisms to visualize distributed systems: a *hierarchical concentric* layout that places containers and entities in a ring of rings, and an *organic* layout that uses the dynamic properties of the system to co-locate entities. We define visualization quality metrics such as intuitiveness, scalability, and genericity, and use them to evaluate the visualization layouts for several application communication topologies including linked lists, trees, hypercubes, and topologies arising from structured overlay networks such as Chord rings.

Index Terms—Graph and network visualization, software visualization, multiple views, scalability issues.

1 Introduction

Distributed systems have recently become very popular, due to their versatility and the relatively low expense of increased computation power through networking. However, these systems are much more complex than their serial counterparts, and thus the burden on programmers and researchers to develop these systems has increased.

For example, consider the problem of observing the computational activity of the computers comprising SETI@home[10], or other applications built upon the BOINC infrastructure[1]. Or, consider visualizing the structure of a BitTorrent[2] swarm as its activity spikes and dwindles over time. Consider a computational scientist who would like to view the interactions between the computers running a high-performance parallel simulation, or a middleware developer who might be interested to verify the locations of various parts of a distributed program, to ensure that it's structure on the network is conducive to efficient inter-process communication. Visualizing autonomous applications, whose reconfiguration is driven by adaptive middleware, is critical to understand the efficiency and stability properties of the middleware's policies.

Since we humans are a fundamentally visually-oriented species, greater productivity and deeper understanding can be given to the designers and maintainers of these systems if there is an available means of visually analyzing distributed systems. Furthermore, effective visualization can also be used as an aid to teach students concepts of distributed systems in an intuitive manner.

In order to properly judge a distributed system visualization's quality, we must first set forth metrics which we can use to more fully gauge its performance.

- **Intuitiveness**: The visualization should provide an intuitive display of both the application-level topology and the physical topology of the network on which it resides. Furthermore, actions should be animated so that it is easy to distinguish changes in the system as they are occurring.
- **Information Content**: The visualization should provide as much insight into the distributed system as it's abstractions permit, unless the user specifies otherwise.
- **Scalability**: The visualization should be as intuitive and intelligible at large scales (for example, with ten thousand distributed system components) as it is in small scales (for example, with thirty).
- **Genericity**: The visualization should be agnostic to the particular type of distributed system in use (for example, it should not matter if the basic computational unit is an actor, a process, or a web service).
- **Interactivity**: The visualization should provide to the user free and intuitive controls, with which they can alter their view of the system at will.

One can immediately sense the difficulty inherent in designing a good visualization: there are essential compromises to be made among different metrics. For example, one must strive to give a high level of information content, yet not clutter the display so as to hamper the visualization's intuitiveness. Well-designed user interfaces should be highly interactive to let users move in the metric space.

In Section 2, we introduce OverView, the toolkit upon which we have constructed the visualizations we examine in this paper. In Section 3, we describe the visualization modules which we have constructed. In Section 4, we examine how these visualizations perform on some common application communication patterns. In Section 5, we evaluate these visualizations in terms of the five visualization quality metrics laid out above, and discuss some of the future work to be done on OverView and the visualization modules described. Finally, in Section 7, we discuss related work.

2 Background

OverView[3][1] is a toolkit which permits visualization of Java-based systems; in particular, distributed systems such as those previously described.[2] The toolkit includes three programs, each of which performs a different task (see Figure 1):

- *Rensselaer Polytechnic Institute, laporj2@rpi.edu.*
- *Rensselaer Polytechnic Institute, cvarela@cs.rpi.edu*

[1] http://wcl.cs.rpi.edu/overview/

[2] OverView versions 0.1 and 0.2 were developed as a plugin for the Eclipse IDE. It has since been made stand-alone, so that it might appeal to those who do not use Eclipse, and so more interesting features might be added: for example, the ability to run the visualization from within a web browser as a Java applet. At the time of writing, the current version of OverView is 0.4.1.

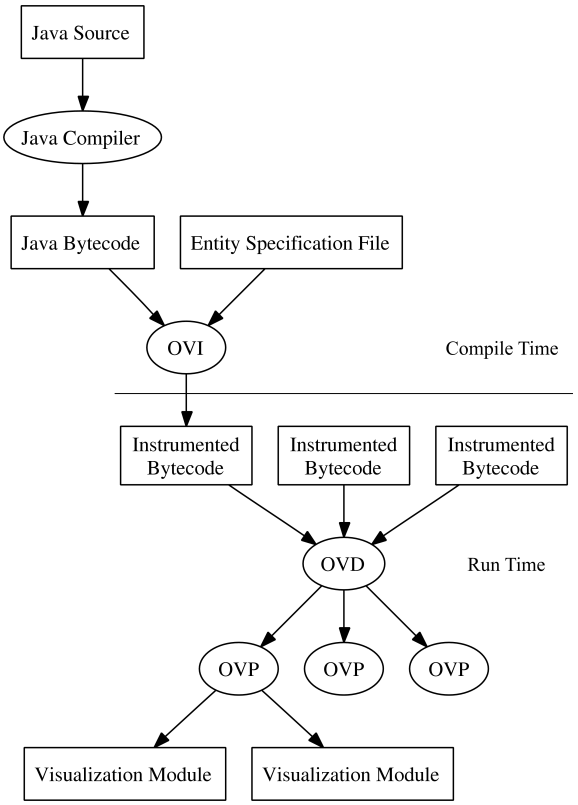

Fig. 1. The OverView framework, showing both compile-time operations and the layout of the run-time architecture.

- The *OverView Instrumenter*, or OVI, which allows the abstraction of a Java program's execution into a set of visualizable events by inserting unobtrusive event-sending behavior into existing Java bytecode.
- The *OverView Presenter*, or OVP, which receives and interprets events into a meaningful, interactive, graphical representation of the state of the distributed system. OVP has several *visualization modules*, each of which can display the distributed system in a different layout.
- The *OverView Daemon*, or OVD, which acts as an event relay, collecting events sent by *event sources* (that is, any active instrumented program), and forwarding those events to *event sinks* (that is, any listening visualization program).

Users add event-sending behavior to any existing Java program by writing an *Entity Specification Language* file, which uses a simple, declarative syntax to map Java method invocations to OverView events. These events are sent, at run-time, over a network to a listening OVD. For a more detailed discussion of OVI's instrumentation and profiling, we refer the reader to [3].

OVP can listen for incoming events from multiple sources, both online network connections and offline log files. It will multiplex these events and forward them to a visualization module, in addition to timestamping and logging them, so that they might be played back at a later date, if desired. OVP maintains an internal log of all events received, providing the ability to "rewind," "pause," and "fast-forward" any visualization at run time. OverView visualization modules are written using the Processing Development Environment,[3] due to it's simple and powerful graphical API, it's support for interactivity, and it's speed compared to other graphical frameworks in Java.

OverView has been used to visualize various distributed environments, most notably by instrumenting the SALSA programming language[11],[4] which is a general-purpose actor-oriented programming language, implemented as an extension to Java. This means that any program written in SALSA will automatically have event-sending behavior simply by using an OverView-instrumented bytecode distribution. Classes of programs which OverView has been used to visualize include parallel iterative and recursive computations.

2.1 Events

OverView's visualization model is based on two fundamental units, called *entities* and *containers*. An entity embodies the concept of a discrete unit of computation, which could refer to an object, an actor, an ambient, a process, or even a virtual machine. A container refers to the environment in which an entity exists; for example, a physical machine (in the case of processes), a virtual machine (in the case of Java objects), or a theater (in the case of actors). Every OverView visualization is composed of some aggregation of these two basic elements.

The events OverView understands take the form of an event type, followed by a variable number of parameters, each of which must be the identifier of some entity or container. OverView visualization modules may each separately define what events they understand; however, the ones we have created understand the following:

- **Position/1**: Tells OverView to create a particular entity, or to move one if it already exists, outside of any container.
- **Position/2**: Tells OverView to create a particular entity inside a particular container. Again, if the entity already exists, it is moved instead of being created. If the container does not yet exist, it is created and placed outside any other container.
- **Deletion/1**: Tells OverView to delete a particular entity. If such an entity does not exist, nothing happens.
- **Communication/2**: Tells OverView that two entities have communicated (for example, via method invocation or message passing).

It is worth noting that some visualization modules do not differentiate between entities and containers. This means that one may recursively nest entities inside of other entities, or delete containers (in which case, all entities within are deleted as well).

3 Visualization Modules

3.1 Hierarchical Concentric

The first module, called *Hierarchical Concentric*, is an example of a visualization in which there is no enforced distinction between entities and containers. All top level entities and containers are arranged in a ring around the center of the screen, each scaling to fit if necessary. Those which are containers are differentiated from entities by being drawn as a square, rather than a circle; furthermore, any entities or containers it contains will be arranged in a circle within it. All interactions that occur feature interpolated animation, using a technique known as *easing*, to animate what is happening. Lines are drawn between communicating entities, which fade over time, but never completely disappear; the benefit of this is to be able to see when two entities are communicating, and also when two entities have communicated in the past, which provides insight

[3] http://www.processing.org/

[4] http://wcl.cs.rpi.edu/salsa/

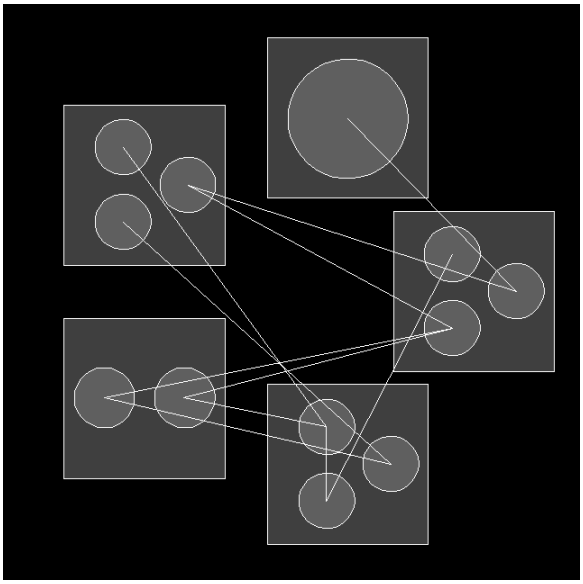

Fig. 2. An example of Hierarchical Concentric, with a random application topology containing twelve entities.

into the application's topology. See Figures 2 and 3 for an example of this visualization. Since there is no enforced distinction between an entity and a container, the semantics of this visualization is such that any entity which contains one or more entities is considered a container. User interactions include being able to control the camera's position and zoom by clicking and dragging the mouse, and displaying the name of an entity or container by hovering over it with the mouse.

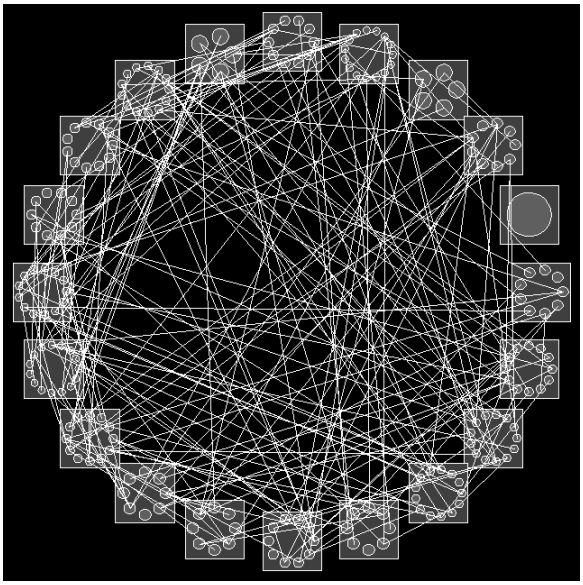

Fig. 3. An example of Hierarchical Concentric, with a random application topology containing two hundred entities.

3.2 Organic

The second visualization module, *Organic*, uses a much more fluid model (see Figures 4 and 5). In this visualization, entities and containers are fully distinct, with only entities being directly drawn, while containers being distinguished by the color of the entity. Entities have no rigid constraints on their position, and float freely on the screen. Two forces act upon each

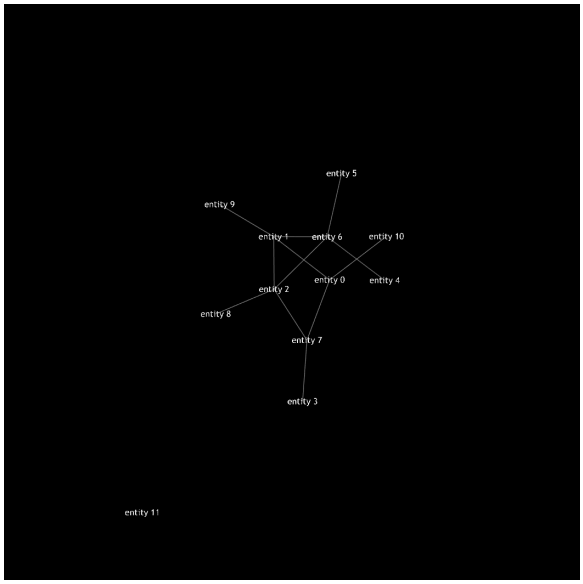

Fig. 4. An example of Organic, with a random application topology containing twelve entities.

entity: first, every entity repels every other entity with a force proportional to the inverse square of distance between them. Secondly, when two entities are connected by a communication, a linear spring force is applied (the magnitude of which scales as communication frequency is increased). Since entities that communicate are likely to be related, these two rules tend to cause related entities to cluster together. Furthermore, since each is colored based on it's container, clustered colors tend to mean that the entities of the distributed system are placed such that there is minimal communication latency, which is a desirable property in distributed systems. Using the Organic visualization, users can click and drag the mouse to change the position of entities.

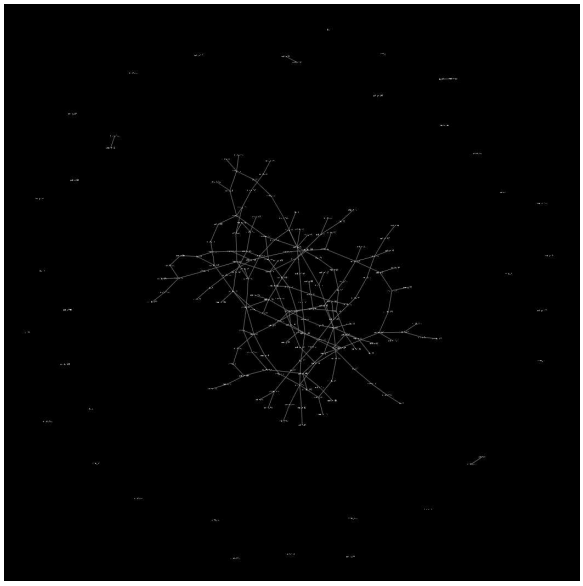

Fig. 5. An example of Organic, with a random application topology containing two hundred entities.

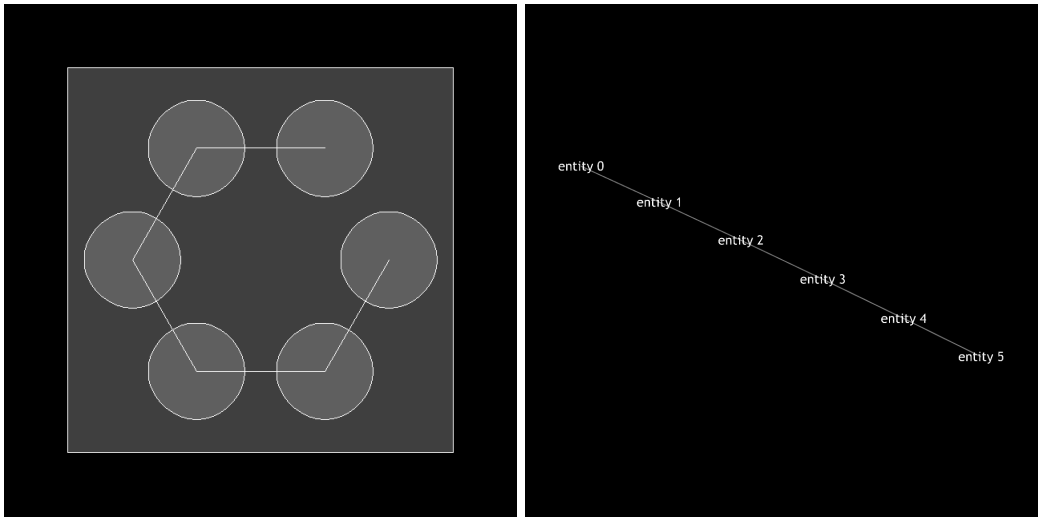

Fig. 6. Example visualization of a linear application topology.

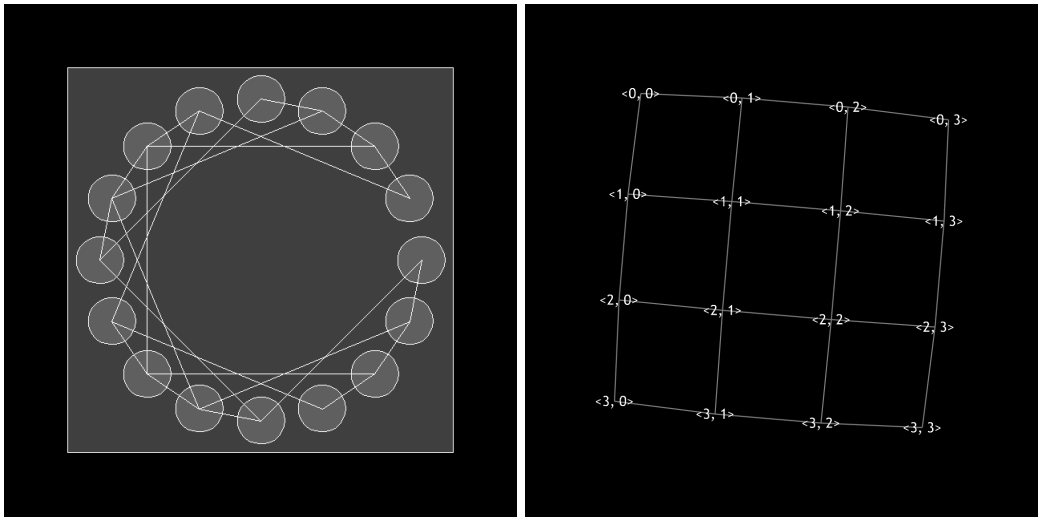

Fig. 7. Example visualization of a grid-based application topology.

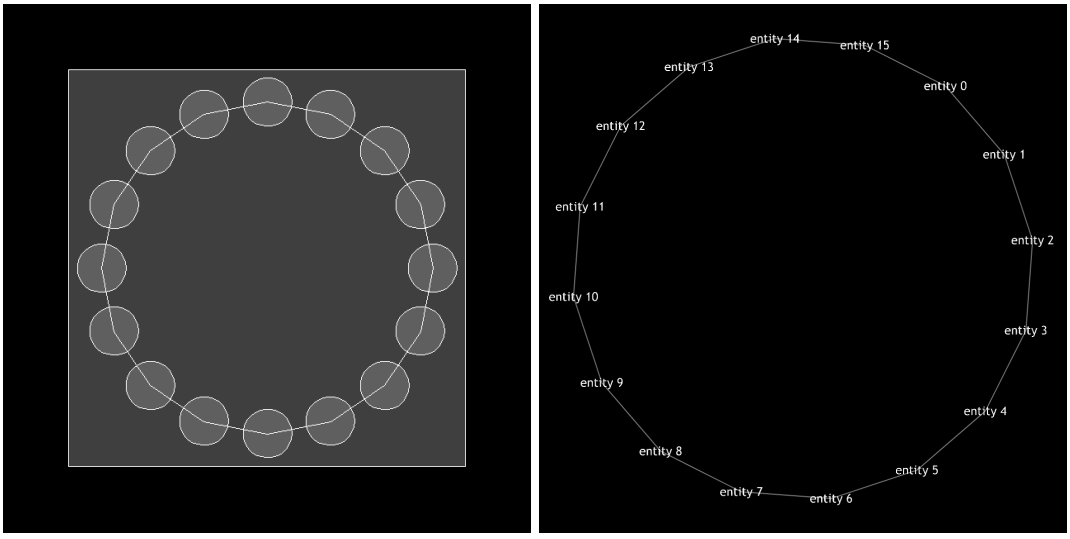

Fig. 8. Example visualization of a ring application topology.

4 Visualizing Common Application Communication Patterns

Herein, we examine the performance of the Hierarchic Concentric and Organic visualizations upon several common application topologies seen in distributed computation.

4.1 Linear- and Grid-based Topologies

Linear- and Grid-based topologies are characterized by having a number of nodes set up in some N-dimensional lattice, with each node connected to and communicating with its neighbors. The most common class of programs that use such a structure are physical simulation programs, such as heat distribution or fluid dynamics applications, where each node represents an area in space.

Figures 6 and 7 show Hierarchical Concentric and Organic visualizations on both a 1-dimensional space and a 2-dimensional space, respectively. Both visualizations are effective on a linear topology, while only Organic is effective on a grid topology. However, for topologies such as these, if it is intelligible at a small scale, it will generally scale indefinitely; and this is true for both visualizations. An exception worth noting is that in the Hierarchical Concentric visualization, the order in which the entities are created is very important. If created in order, they will appear as in Figure 6; if they are created in an arbitrary or random order, however, the lines denoting their connection to their neighbors will criss-cross across the ring, and become quite difficult to understand. However, regardless of creation order for Organic, the visualization will settle into the configuration shown in Figures 6 and 7.

4.2 Ring-based Topologies

Ring-based topologies find a common application in peer-to-peer networks. This is because they are easy to construct with only local knowledge, and information queries, while not necessarily efficient, are very simple (usually as simple as merely passing queries around the ring in one direction until they are answered). As an optimization upon this, Chord[9] increases the number of neighbor links from a constant number to a logarithmic number (based on the size of the network), which decreases the search time from linear to logarithmic as well; this makes it possible to construct simple and high-performance peer-to-peer networks.

Figures 8 and 9 show how the two visualizations perform on these structures. While, once again, these topologies scale well (for all except Organic on the Chord ring), the Hierarchical Concentric visualization is very dependent on the order in which the entities are created. Interestingly, while Organic usually fares well under many application topologies, the communication pattern for Chord is sufficiently dense such that it compacts together heavily, becoming unreadable.

4.3 Hypercube-based Topologies

Cubes, hypercubes, and other N-cubes have a number of notable properties useful to distributed computation; among them is the upper bound on hops between any two nodes on the network, while maintaining only a small number of edges between nodes.

Hypercubes are also notable for having a very structured 4-dimensional organization, which is difficult to map to 2-dimensions well. This can be seen in Hierarchical Concentric, since while it is intelligible, it is not at all intuitive. However, Organic performs well on the hypercube, placing nodes in a well-spaced layout (see Figure 10). Unfortunately, though, neither perform well at larger scales, as the increasing number of dimensions makes it increasingly complex to understand.

4.4 Recursive Topologies

Recursive computation is as important in distributed computation as it is elsewhere; many divide-and-conquer recursive algorithms can be made to run in a distributed setting.

When nodes are created with a particular ordering, the Hierarchical Concentric visualization can perform reasonably well on binary trees; fortunately, this is not uncommon with recursive algorithms. However, even on top of this, Organic performs excellently with trees, which tend to balance well and have a clear representation, with the root generally centered and branches evenly spaced apart. Figure 11 uses the example of the recursive computation of the sixth number in the Fibonacci sequence; noting its fractal nature in Organic is particularly simple.

5 Discussion and Future Work

5.1 Visualization Modules

Hierarchical Concentric was designed as a straightforward refinement upon previous visualizations, which laid out entities and containers in a simplistic grid fashion—left-to-right, and top-to-bottom. Instead of this, Hierarchical Concentric places entities around a hierarchical series of rings. This placement mechanism was selected for several reasons: firstly, it minimizes ambiguous communications—that is, communication lines overlapping multiple entities, so one cannot determine at a glance which entities are referred to; this is a commonly-arising problem in grid-based visualizations, since many communications will either be purely horizontal or purely vertical and reach across entire rows or columns. Secondly, it can display a hierarchy of elements intuitively, even with naïve placement of new elements at the end of the list, allowing it to support recursive computations with reasonable effectiveness. This hierarchical, concentric structure is both intuitive and generic, mapping naturally both to flat systems (such as actors within theaters) and nested systems (such as mobile ambients).

In Hierarchical Concentric, when two entities communicate, a simple line is drawn between them, which fades over time. While this tells one of instantaneous communication information, most of the historical communication information—other than knowing the binary fact of whether two entities have communicated or not—is lost. This causes the visualization to fall somewhat short of the amount of information content it could display.

Hierarchical Concentric's main drawback is in scalability. While it is effective for small numbers of entities, it rapidly breaks down and becomes confusing with as few as one hundred entities, with "spider-webbing" communications filling the rings. Furthermore, logically grouped entities may be placed far from each other by the visualization, and since one cannot zoom in upon both sides of the ring at once, it can become difficult or impossible to understand what is occurring in parts of a system.

However, despite these weaknesses, the visualization can be very effective for some application topologies, such as linear and circular topologies, as demonstrated above.

Organic was primarily developed to remedy some of the shortcomings of Hierarchical Concentric. Unlike it, there is no hierarchical placement; entities are only placed upon the primary visualization surface, and cannot be nested. It uses only two simple placement rules; however, these two rules provide a surprisingly effective visualization, causing application topologies to emerge in clear patterns. Furthermore, if like colors are well-clustered, it demonstrates that entities that communicate frequently are near each other physically, which is a very natural notion for the visualization to convey. We note, therefore, that Organic provides a very intuitive visualization, especially in reference to the application topology.

Since Organic's placement algorithm is based on local rules, it may not always find the best placement scheme. However, users may click and drag objects to place them wherever they wish. This is simple, intuitive, and easily remedies the system's placement weaknesses.

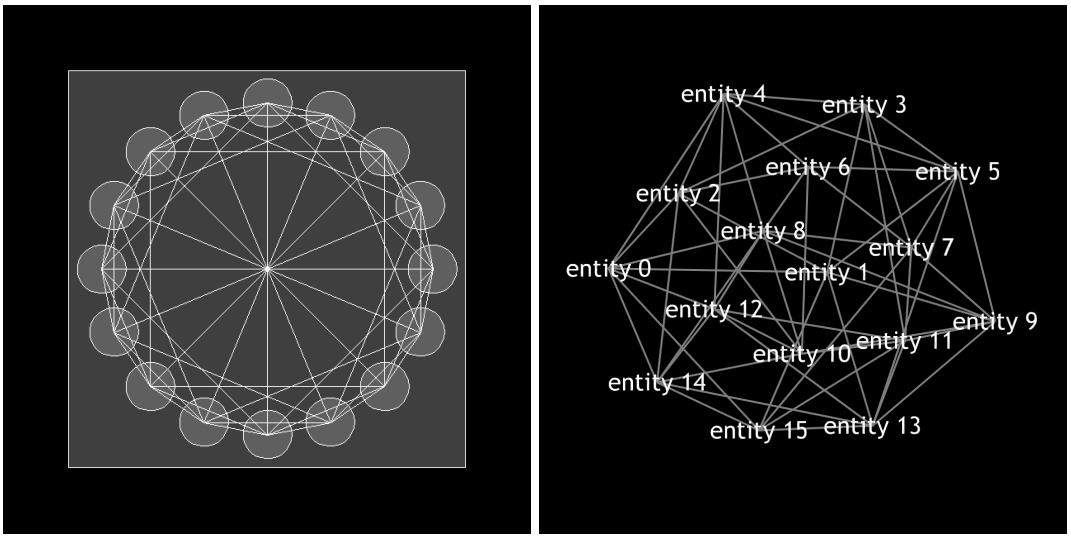

Fig. 9. Example visualization of a Chord ring.

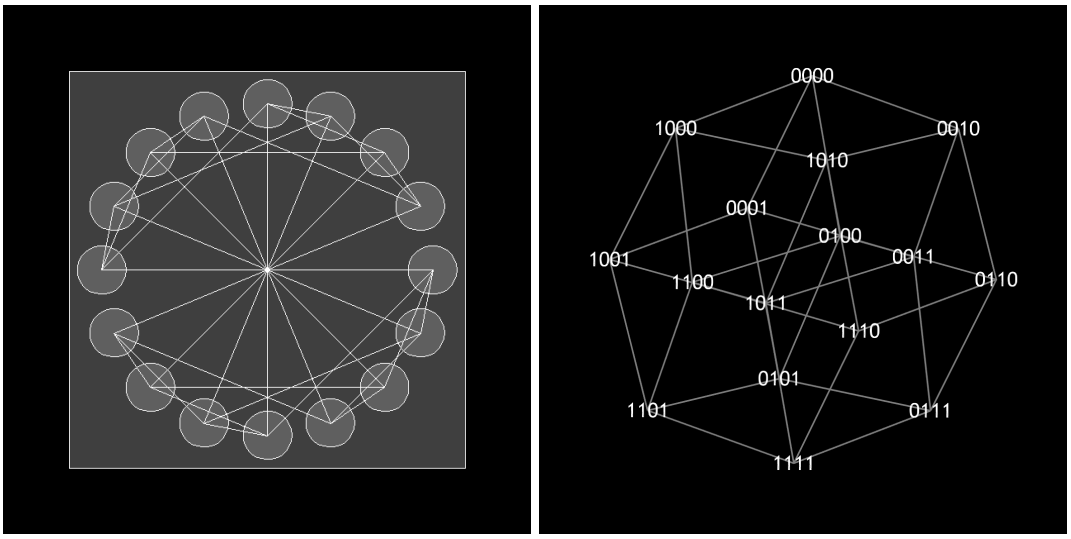

Fig. 10. Example visualization of a hypercube application topology.

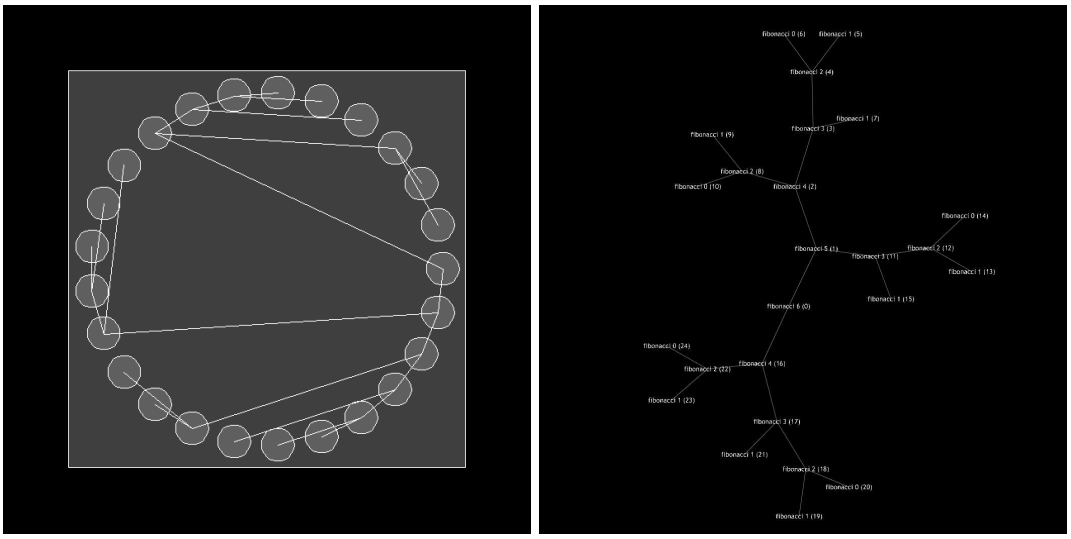

Fig. 11. Example visualization of a recursive application topology.

		H. C.	Organic
Intuitiveness	Linear	⋆ ⋆	⋆ ⋆ ⋆
	Grid		⋆ ⋆ ⋆
	Ring	⋆ ⋆	⋆ ⋆ ⋆
	Chord	⋆ ⋆	
	Hypercube		⋆ ⋆
	Recursive	⋆	⋆ ⋆
Information Content		⋆	⋆ ⋆
Scalability	Linear	⋆ ⋆	⋆ ⋆ ⋆
	Grid		⋆ ⋆ ⋆
	Ring	⋆ ⋆	⋆ ⋆ ⋆
	Chord	⋆ ⋆	
	Hypercube		⋆
	Recursive	⋆	⋆ ⋆
Genericity		⋆ ⋆ ⋆	⋆ ⋆
Interactivity		⋆	⋆ ⋆

Fig. 12. A table, referencing the relative effectiveness of each visualization with each application topology, along each metric laid out in the introduction. The scale goes from zero stars, for poor, to three stars, for excellent.

Organic is less generic than Hierarchical Concentric, lacking support for nested distributed systems. However, balancing this, it also scales much better than Hierarchical Concentric does, able to handle several hundred entities and remain comprehensible. Finally, since repeated communication pulls entities closer together, one can also get a sense of how tightly two entities are coupled, preserving frequency of communication information, giving somewhat more information content than Hierarchical Concentric.

Our experiences with these two visualizations thus demonstrate that neither is a panacea; while we developed Organic to be more effective than Hierarchical Concentric, it is only demonstrably so in particular cases, as Figure 12 illustrates.

5.2 Future Work

While the development of a better visualization is an open-ended process, there are several directions in which we are currently working.

The most important short-term goal which we are striving towards is scalability; both in visualization and in event-transmission framework. Organic is our current attempt at moving towards more scalable visualizations; while Ring can only present a few dozen entities effectively, Organic can effectively present a few hundred or more; techniques learned include designing for adaptivity (hard-coded schemes do not scale well), and that interactivity is key (allowing a user to move objects around allows them to fix any problems the visualization is unable to on its own).

Concerning making a more scalable framework, we are currently focused on the development of *event filters*, which will allow one to filter out unwanted messages at run time. By pushing these event filters from the event sinks to the OVD relays, and indeed, as close to the event sources as possible, we can reduce the number of events that need to be sent, which makes better use of network bandwidth; more importantly still, it can reduce the amount of information to be visualized, which can simplify scaling up the visualization. Since multiple event sinks can exist on the network, there must be a mechanism to determine the smallest set of events which need to be forwarded, based on the different requirements of the various filters. However, this is not an insurmountable problem, and with careful selection of filters, we expect to be able to reduce visualization network traffic and visual clutter markedly.

Finally, we are also interested in developing a generic data structure which contains all information necessary for visualization modules to render a visualization. The motivation for this is that, currently, different visualization modules may only be selected starting an execution; it is important to be able to toggle between various visualization modules as the visualization is running.

6 Related Work

Frishman and Tal[4] have developed a visualization tool which bears a number of similarities to our own. While it is limited to using mobile objects (a single model of distributed systems, and thus is less generic), it takes an interesting approach to scalability. It suggests providing the user a means to select one or more points or focus; that is, parts of the visualization that are of interest. Then, the software visualization will use an algorithm to determine which mobile objects it can filter out as being uninteresting to the user. This stands in contrast to its negative: the approach of selecting the objects one wishes to filter out. Such a means might provide a better-scaling visualization; ultimately, there will always be fewer entities one is interested in than entities one is not.

Another software visualization framework which is similar to OverView is EVolve[12], having the notion of events which describe the runtime behavior of an application being translated into a graphical visualization. It focuses upon object-oriented computations, supporting visualization of method invocations; while it's visualizations are less intuitive and generic, they are scalable and provide a large amount of information content, and may even be layered naturally on top of each other to show the user different types of information at once.

Benjamin Fry's *Organic Information Design*[5] demonstrates that while static visualization systems are well understood, due to their long history, dynamic visualization systems made possible by computers 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, scalable, and intuitive visualization. The downside to organic information design is that it represents data in a qualitative, rather than quantitative fashion. However, qualitative visualizations are preferable at large scales. He also demonstrates examples of his visualizations on very large scale systems, such as the human genome. By taking cues from other large-scale, dynamic systems, we might better be able to visualize large-scale distributed systems as well, which are defined by their highly-dynamic nature.

Orla Greevy *et al.*[6] describe a three-dimensional approach to software visualization, placing objects in the horizontal plane, as we do, but using depth to convey dynamic runtime information, such as object instantiation and message passing.

Work describing the effective layout of hierarchical graphs, which directly relate to the visualization of distributed systems, include Keskin and Vogelmann[7], who use the visual metaphor of the *cityscape* to create a dense, but intuitive representation of hierarchical graphs, and Kusnadi *et al.*[8], who make use of neural networks to decide the the layout of the graph.

7 Conclusion

Distributed systems are prevalent today and are continually becoming even more prevalent; due to their complexity, effective visualization is necessary. In the effort of examining these visualizations so as to build more effective ones, we set forth visualization metrics, which aid in more completely analyzing a visualization.

OverView is a tool for visualizing distributed systems in Java, with a modular design allowing for inserting new visualizations into it. OverView visualizations are based on the notions of entities, or computational units, and containers for these entities. We have developed two distinct visualizations modules for OverView, *Hierarchical Concentric* and *Organic*, and have evaluated these modules in terms of the metrics which

we have specified. Hierarchical Concentric uses a rigid, ring-based placement mechanism and excels at visualizing linear application communication patterns, but does not scale well. Organic uses a free-form, rule-based placement mechanism which is effective for visualizing many application topologies, and scales well; however, it is incapable of visualizing systems composed of nested entities. We derive from our experimentation with these visualization systems that local, emergent entity placement rules are beneficial in terms of intuitivity and scalability, and by including a high degree of interactivity, many of the visualization difficulties that can arise from the simplicity of these rules can be mitigated by the user.

Dynamic visualizations and distributed systems are both very recent developments which are in their infancy; there is much interesting work yet to be done, and the development of better visualizations for distributed systems is an ongoing one. We are particularly interested in developing the scalability of these visualizations, from hundreds of entities to thousands or tens of thousands of entities; the progress made thus far is a solid step in the direction of that ultimate goal.

References

[1] D. P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing.*, pages 4–10. IEEE Computer Society, 2004.

[2] B. Cohen. Incentives build robustness in BitTorrent. Technical report, May 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] Y. Frishman and A. 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.

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

[6] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3d. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2006. ACM Press.

[7] C. Keskin and V. Vogelmann. Effective visualization of hierarchical graphs with the cityscape metaphor. In *NPIV '97: Proceedings of the 1997 workshop on New paradigms in information visualization and manipulation*, pages 52–57, New York, NY, USA, 1997. ACM Press.

[8] Kusnadi, J. Carothers, and F. Chow. Hierarchical graph visualization using neural networks, May 1997.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[10] 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 computers. 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–+, Jan. 1997.

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

[12] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge. Evolve: an open extensible software visualization framework. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 37–ff, New York, NY, USA, 2003. ACM Press.