Visualization of Mobile Object Environments

Yaniv Frishman[*]
Department of Computer Science
Technion - Israel Institute of Technology

Ayellet Tal[†]
Department of Electrical Engineering
Technion - Israel Institute of Technology

Abstract

This paper presents a system for visualizing mobile object frameworks. In such frameworks, the objects can migrate to remote hosts, along with their state and behavior, while the application is running. An innovative graph-based visualization is used to depict the physical and the logical connections in the distributed object network. Scalability is achieved by using a focus+context technique jointly with a user-steered clustering algorithm. In addition, an event synchronization model for mobile objects is presented. The system has been applied to visualizing several mobile object applications.

CR Categories: I.3.8 [Computer graphics]: Applications; H.5.2 [Information interfaces and presentation]: User Interfaces—Graphical user interfaces; C.2.4 [Computer-communication networks]: Distributed systems—Distributed applications

Keywords: Software visualization, distributed software visualization, mobile objects

1 Introduction

Software visualization is concerned with the creation of tools and techniques to visualize different stages of software and algorithm development [Stasko et al. 1998]. As software systems become more complex, the need for effective methods of providing insight into the structure and behavior of programs, increases.

As a consequence of the improvement in communication technology and the growth of the Internet, the use of distributed software is becoming more widespread. The mobile object paradigm, a recently introduced architecture for distributed computing, allows programs to migrate to remote hosts while they are running [Acharya et al. 1996; Holder et al. 1999a; Holder et al. 1999b; Jum 1999; Milojicic et al. 1999; Voy 1997; Walsh et al. 2000]. This paradigm offers scalability, availability and flexibility advantages compared to other methods of creating distributed applications. Creating effective visualizations of mobile object frameworks is a challenging problem and is the subject of this paper.

Several tools have been developed for visualizing parallel and distributed programs [Kraemer and Stasko 1993]. The PVanim system [Topol et al. 1998] is a toolkit for creating visualizations of the execution of PVM programs. PARADE [Stasko and Kraemer 1993] in an environment for developing visualizations of parallel and distributed programs. In [Moe and Carr 2001], tracing of

[*]e-mail: frishman@tx.technion.ac.il
[†]e-mail:ayellet@ee.technion.ac.il

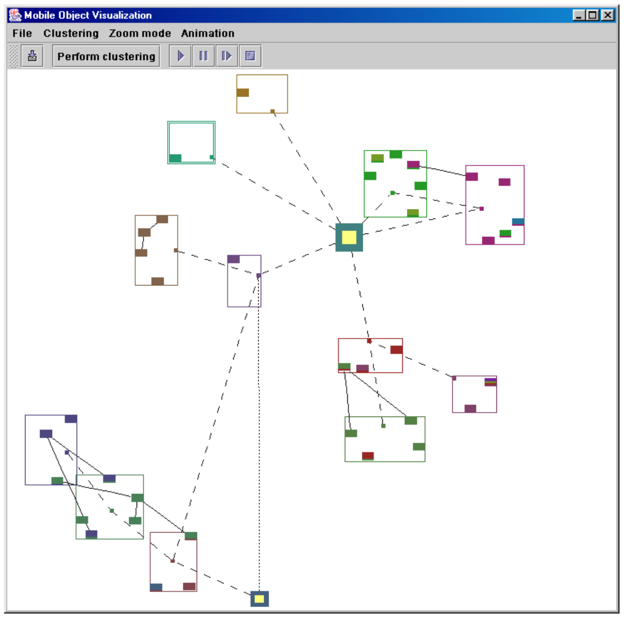


Figure 1: Visualization system GUI

CORBA [OMG 1998] remote procedure calls is used to analyze runtime activities and look for anomalous behavior. Vade [Moses et al. 2004] is a distributed algorithm animation system in which visualizations can be created and executed on a web page on the client's machine. Pablo [Reed et al. 1993] provides analysis and presentation of performance data for massively parallel distributed memory systems. Jinsight [Pauw et al. 2001] is a system for the visual exploration of the run-time behavior of complex Java programs.

Although research on mobile objects is widespread, visualization of such frameworks has hardly been done. To our knowledge, the only work in this field is [Wang and Kunz 2000], where a modification of the process-time diagram, adopted from XPVM [Kohl and Geist 1996], is used as the means of visualization. The creation, destruction and movements of mobile objects are visualized. Event synchronization is handled by timestamps and ordering rules. This system has a few drawbacks. It requires manual annotation of source code in order to generate events. It does not visualize the communication between objects nor does it display information regarding the communication between machines. The system does not visualize the physical connections between machines nor does it display the logical connections between objects. Finally, it is not scalable. In this paper we discuss a different approach to the visualization of mobile object frameworks, attempting to solve these problems.

This paper makes the following contributions: First, we detail the requirements from a visualization system for mobile objects. Second, a graph-based visualization that concurrently shows the physical connections in the computer network as well as the logical relations between the mobile objects is presented (see Figure 1). Third, A context-sensitive focus+context fisheye type display technique is suggested in order to provide hierarchical information display and support scalability. Fourth, A clustering algorithm, which is affected by nodes of interest to the user, is presented. Fifth, we present a model for event synchronization that is used to guarantee visualization consistency. Finally, we propose a method in which events are automatically generated, avoiding additional work by the programmer of the application.

The rest of this paper is structured as follows: Section 2 provides the background on mobile object frameworks. In Section 3, the requirements from a mobile object visualization system are discussed. Our visualization is presented in Section 4. Section 5 describes the methods developed in order to make the visualization scalable. Section 6 discusses implementation issues, in particular event generation and synchronization. A case study is described in Section 7. Finally, Section 8 concludes and discusses future directions.

2 Mobile Objects

In recent years, distributed objects have become prominent in the design of distributed applications [Brown and Kindel 1998; OMG 1998; Sun 1997]. Mobile objects are a natural evolution of the distributed objects concept [Acharya et al. 1996; Holder et al. 1999a; Holder et al. 1999b; Jum 1999; Milojicic et al. 1999; Voy 1997; Walsh et al. 2000]. A mobile object framework has a few distinctive features. The first feature is *code mobility*: objects can migrate to remote hosts, together with their state and behavior, while the application is running. We refer to the processes hosting mobile objects as *cores*. Mobility may be further classified into *weak mobility*, where only the object's code and state are moved, as opposed to *strong mobility* where the full runtime context, including the stack and the program counter, are moved.

The second feature of mobile objects is *location transparency*, which allows the programmer to make calls to objects regardless of their current location. Since the location of objects may change over time, provisions must be supplied in order to track referenced objects. Unlike regular distributed objects, in which the location of a remote object is fixed, when making a call using a reference to a mobile object, the parameters may pass through several intermediary cores until reaching the called object. The introduction of intermediary cores allows for a more scalable, lazy update of the location of a referenced object [Holder et al. 1999a].

Another feature of mobile object frameworks is *layout programming*. Apart from the behavior of objects, their co- and re-location constraints may be specified. For example, the programmer may specify that two objects α and β should always be located at the same machine. This may be necessary if α and β are expected to perform a lot of communication, for example.

In order to make effective use of mobility, a *monitoring facility* is used to get up to date information about the network status and object deployment. This profiling infrastructure can be part of the mobile object infrastructure [Gazit 2000] or an external process [Acharya et al. 1996].

Mobile object frameworks have several benefits. The first is flexibility. In cases where a great deal of communication is expected to be performed between two remote objects, α and β, α can migrate, along with its state and behavior to β's location and perform the communication locally, thereby avoiding the long delays associated with remote method calls. If α needs to interact frequently or pass a large amount of data to β, it might be preferable to pay the overhead associated with migration rather than the cost of remote communication.

The second benefit of mobile objects is their easy deployment: assuming a widespread existence of appropriate middleware, it is possible to deploy applications to new cores without the need to manually install them ahead of time.

Third, mobile objects are scalable. When deploying an application on a large number of cores, it is often impossible to know a priori the optimal network layout the application should use in order to best leverage the resources of the computer network. The constantly changing availability of communication and computing resources call for the flexibility of dynamic object relocation in order to maximize the use of available resources.

Finally, movement may also be used to increase the availability of the application: in cases of failure of parts of the network, the application can migrate or regenerate on live parts of the network.

3 Requirements

This section discusses the requirements from a visualization system for mobile objects.

1. Physical and logical visualization: A mobile object application has two distinct, yet related facets. The first is the physical computer network with the interconnections between the cores. The second is the logical network of mobile objects that can be used to show the connections and interactions between objects. The visualization should display both of these facets.

2. Interesting events: In any visualization system, the events that need to be visualized greatly affect the design of the system. In the case of mobile objects, the following interesting events should be visualized:

- Object Movement: The movement of objects between cores while the application is running is the main difference between mobile object frameworks and regular distributed applications. Therefore, a clear and concise depiction of such activities is of great importance.
- Construction/destruction: Being a dynamic, distributed application, both objects and cores may be added or removed during the execution of the application.
- Communication: Being distributed in nature, the messages sent between the different parts of the system play a paramount role during execution of the application and therefore provisions to visualize them should be supplied.

3. Consistent depiction: A consistent depiction of the events occurring in the system is of vital importance. This is a challenging requirement since in distributed environments there is no global clock that can be used to synchronize the events emitted by different processes. Visualizing the execution of mobile objects adds an additional complexity because parts of the application change their physical location.

4. Transparency: Event generation should be transparent both to the programmer and to the user of the application. Moreover, care must be taken in order to reduce the perturbation of the application caused by generating the events.

5. Interactive control: The user should be able to steer the visualization system to display relevant and interesting data out of the large amount of information collected. This control should be interactive, allowing the user to feed back to the system new requests based on the knowledge accumulated while viewing the unfolding visualization.

6. Scalability: One of the main challenges in software visualization is building a scalable visualization. This is especially important when dealing with networks of computers, which can potentially generate massive amounts of information. A visualization system should be able to process large amounts of data while avoiding swamping the user with information.

7. Online and offline: In order to facilitate both online and offline (post-mortem) inspection of application execution, provisions should be supplied in order to record and playback a sequence of captured events. This capability allows, for example, the comparison of two different runs of the application, for debugging or optimization purposes.

The following sections address these requirements. Section 4 discusses requirements 1 and 2. Section 5 addresses requirements 5 and 6. Section 6 addresses requirements 3, 4 and 7.

4 Mobile Object Visualization

As discussed in Section 3, two simultaneous networks are of interest: the physical network of cores (machines) and the logical relations and interactions between mobile objects. A graph is a natural choice for visualizing a distributed network. In our case, we need to simultaneously visualize two graphs. We use the following definition:

Definition 4.1 ***Clustered Graph:*** *A clustered graph is an ordered quadruple $G = (V, C, E_v, E_c)$, where V is the node set, C is a set of clusters which form a partition of the node set V, E_v is the set of edges between nodes $E_v \subseteq \{ (v_i, v_j) | i \neq j, v_i, v_j \in V \}$ and E_c is the set of cluster-cluster edges $E_c \subseteq \{ (C_i, C_j) | i \neq j, C_i, C_j \in C \}$.*

A clustered graph is a natural choice for displaying the simultaneous physical and logical graphs, as demonstrated in Figure 1. Every mobile object is depicted by a node in the graph. The logical connections between objects are shown using solid edges connecting the nodes. In order to overlay the physical structure of the network, clusters are used. Each core is represented by a cluster that contains all of the objects currently residing in that core. Dashed cluster-cluster edges are used to represent physical connections between cores (see Figure 5), as opposed to logical relations that exist between objects.

The graph can be displayed in three dimensions, as illustrated in Figure 2. Edges between nodes, showing relations between mobile objects, are drawn on the lower plane, while cluster-cluster edges, showing physical connections between cores, are drawn on the upper plane. In 3D, a cluster is drawn as a semi-transparent pyramid. A small dummy node is added to each cluster, drawn at the apex of the pyramid and serves as the endpoint of cluster-cluster edges. One of our guidelines in creating this visualization is being able to collapse the 3D view into a 2D view in a natural and comprehensible way, as illustrated in Figure 3, which shows a 2D drawing of the graph from Figure 2.

We use several techniques and attributes in order to display information in this graph. Each cluster boundary is drawn using a different color. This helps the user track the different clusters while changes are performed to the graph during the visualization.

Each node is drawn using color strips, utilizing the *growing squares* [Elmqvist and Tsigas 2003] metaphor, as shown in Figure 4. The strips are colored according to the location history of

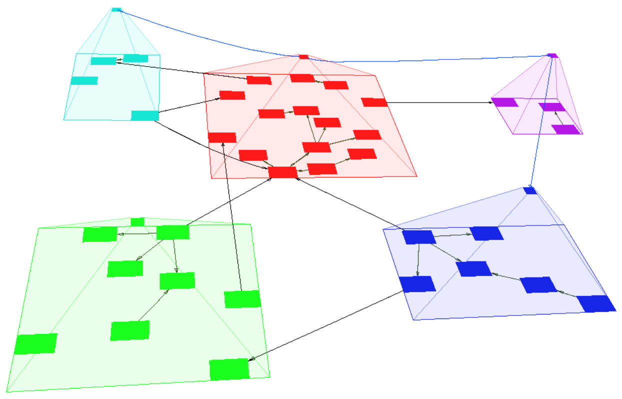


Figure 2: 3D view of a mobile object environment

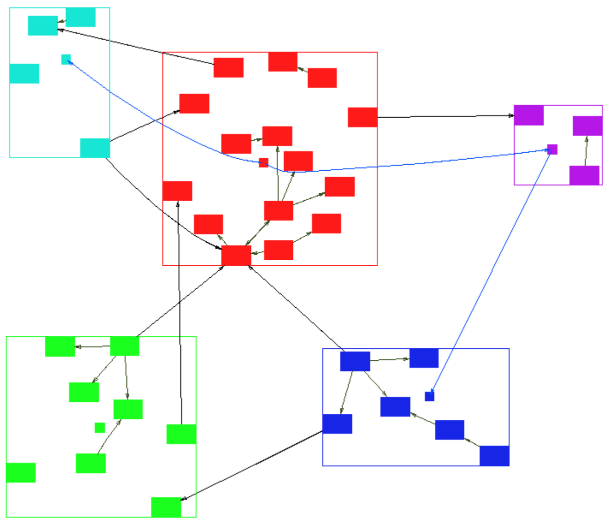


Figure 3: 2D view of a mobile object environment

the object. The bottom strip is the current location (e.g. colored with the same color as the cluster the node currently resides in), the strip above corresponds to the previous location, etc. The maximum number of strips is configurable. Using the color strips, the user can easily distinguish between highly mobile objects (showing strips with many colors) and static ones (showing a single strip).

In order to create a more scalable and meaningful display, we employ lazy construction of edges. A new edge is drawn between two nodes once a method call between the objects is detected. This avoids cluttering the graph with edges (references) that are not used. It may be argued that such unused references are interesting, however, the information gained is usually not worth the extra cluttering of the graph or the increased complexity in detecting these references.

In addition to the existence of communication between objects or cores, the frequency of this communication is of interest to the user. Line patterns are used to convey this information. The higher the frequency of alternation in the dashed lines, the higher the frequency of communication. See for example Figure 1. The sum of two weighted averages is used to calculate the amount of communication between cores. The first is the average number of objects moving between the cores connected by the edge. The second is the average number of remote invocations performed between the

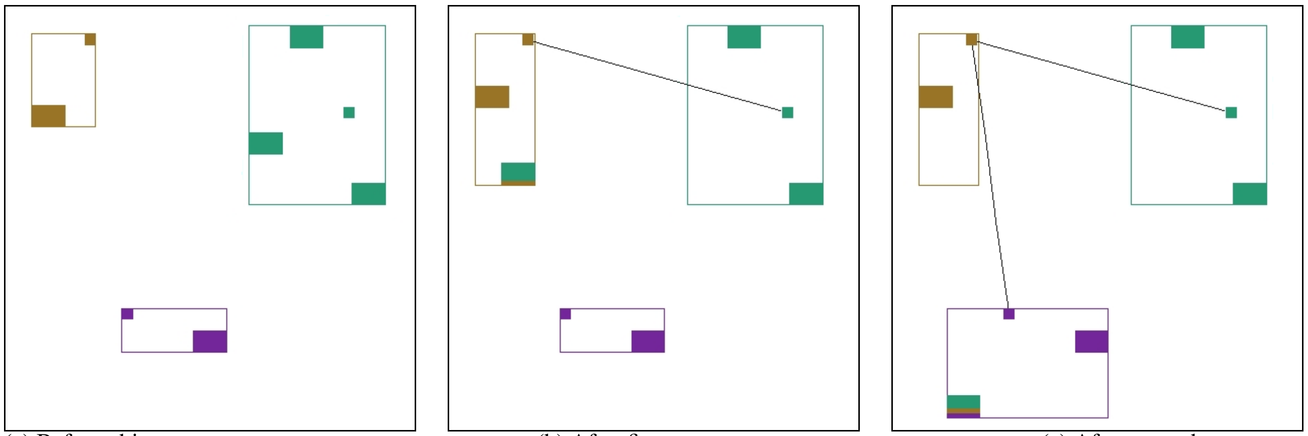

(a) Before object movement (b) After first movement (c) After second movement

Figure 4: Using color to show location history

two cores. The averages are calculated using a weighted sliding window, taking the last *N* samples into account.

In some cases, it is more revealing to use icons, rather than rectangles, to represent the objects. Our system allows the user to assign icons to the nodes, as illustrated in Figures 9 and 10. Using information about the type of each displayed object and a user-defined mapping between texture files and object types, the display can be augmented with class information.

Some mobile object frameworks [Holder et al. 1999a] allow tagging of specific objects as stable, i.e. objects that remain at the same location throughout their lifetime. Such objects may be used, for example, to provide abstractions for hardware devices such as printers or scanners connected to a specific computer. This distinction between stable and movable objects is visualized by laying out the objects in each cluster using two concentric circles. The inner circle contains the stable objects while the outer one contains movable ones.

We have developed a special incremental graph layout algorithm tailored for the requirements of mobile object visualization [Frishman and Tal 2004]. The algorithm produces a dynamic display of clustered graphs, attempting to preserve the users mental map of the graph, as it is being changed [Misue et al. 1995; North 1995]. The algorithm uses a static force-directed layout algorithm as a basic building block [Ellson et al. 2002; Kamada and Kawai 1989; Tollis et al. 1999]. It uses invisible *dummy nodes* to create the clustered structure and *place-holder nodes* to maintain layout stability. Edge length and weight are used as a means of controlling the changes made to the layout.

Our system uses animation in order to show different events. When a new graph layout is performed, for example after an object moves from one core to the other, the positions of nodes, edges and clusters are linearly interpolated between the old and the new locations. A method call between two remote objects is animated using a lightning bolt icon that moves from the caller to the called object.

5 Scalable Visualization

As the number of objects and cores increases, the visualization might get cluttered with information. Gaining any insight from the visualization will become increasingly difficult. In this section we present a context sensitive focus + context technique that alleviates this problem.

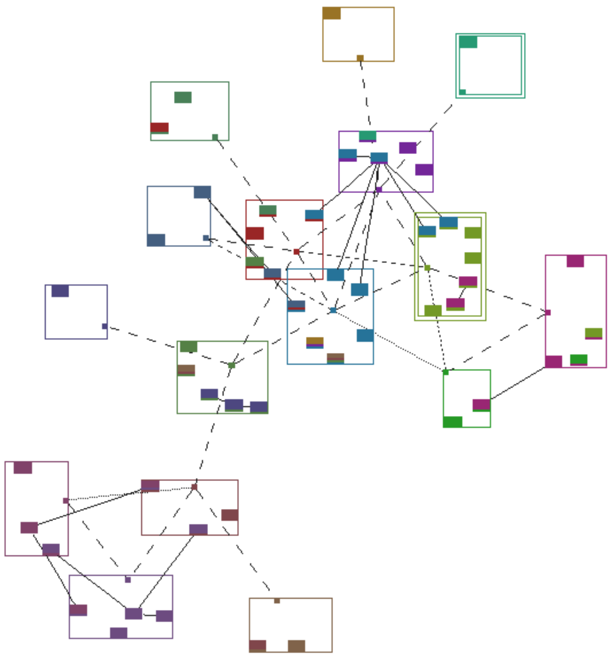
5.1 Levels of Detail

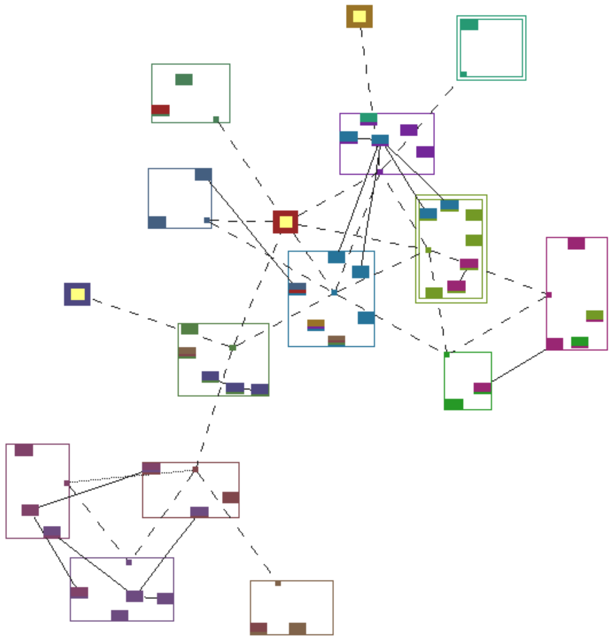
As the simultaneous core/object graph, presented in Section 4, grows larger, it becomes difficult to display all of the nodes and edges in the graph at the same time. The visualization should provide the user with an overview of the graph while at the same time allowing focusing on specific, user-defined areas in order to get more detailed information [Card et al. 1999; Furnas 1986]. To achieve these goals, a hierarchy of levels of detail is defined. Different parts of the graph can be displayed in different levels of detail.

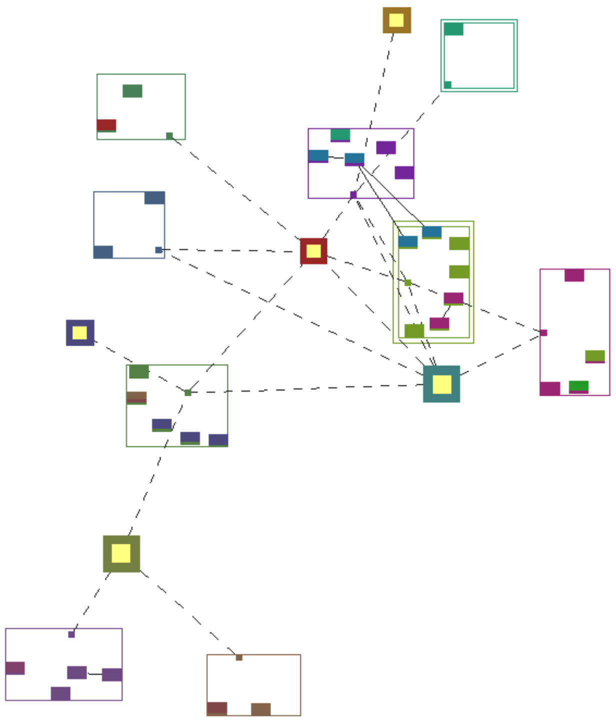
At the highest level, full information is displayed, as shown in Figure 5(a). The next level of the hierarchy omits information about the objects residing on each core. Instead of displaying a cluster for each core in the network, a single node is used to depict each core. As before, edges are used to convey the physical connections to other cores in the network, as demonstrated in Figure 5(b). The final level of the hierarchy combines several cores into one node in the display. The size of such a node is proportional to the number of cores it depicts. Figures 5(c) and (d) demonstrate graphs containing nodes of various levels of detail.

The user has several methods to control which parts of the graph will be displayed in which level of detail. The first is selecting focal nodes (cores) that are of primal interest to the user and thus should be displayed with full detail. The second method is navigating the graph using zoom-in and zoom-out operations. The third is choosing the total number of nodes to be displayed in the graph and letting the system cluster the graph nodes accordingly. Once the user selects focus nodes, a clustering algorithm is employed in order to decide in what level of detail each core will be displayed, as described in the next subsection. The different levels of detail are combined in a seamless, dynamic, user-steered manner.

Zoom–in and zoom–out operations are animated smoothly. The old nodes fade out of the graph while the new nodes fade in. Next, the new nodes smoothly move to their final location. This helps the user understand the changes to the graph. A similar animation is performed when re–clustering is performed. The locations of the new clusters are calculated by the layout algorithm, which takes into account the previous locations of the nodes comprising the cluster, thus maintaining layout stability.


(a) Original graph – 16 clusters


(b) Manually zooming out of 3 clusters


(c) Clustering to 14 clusters

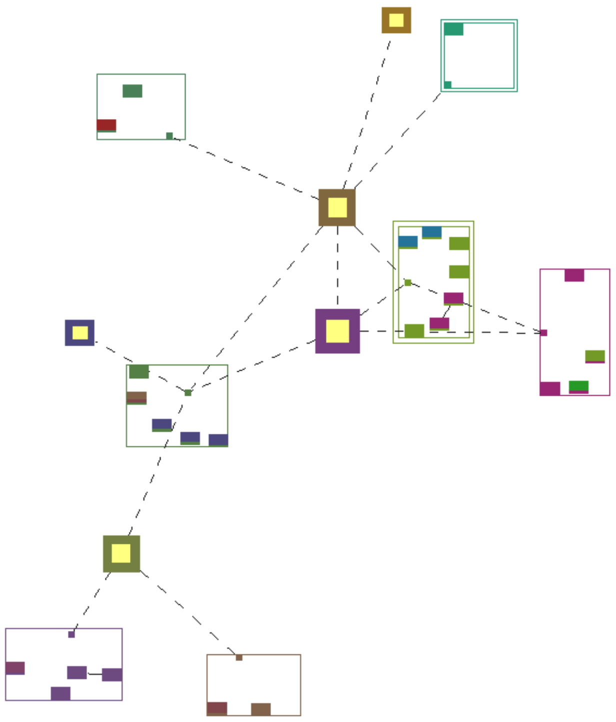

(d) Clustering to 12 clusters

Figure 5: Levels of detail

5.2 Clustering

In this subsection we present the clustering algorithm that computes the focus-based, hierarchical representation of the graph. The algorithm, which is summarized in Figure 6, is based on an extension of the agglomerative clustering algorithm [Duda et al. 2000].

Input: Set of focal nodes; distances between nodes; number of desired clusters
Algorithm:
1. Calculate shortest distance between each node and the closest focus node.
2. Update distances between nodes according to distance to focus node.
3. Perform hierarchical clustering.

Output: Clustering hierarchy of the nodes

Figure 6: Focus-based clustering algorithm

The algorithm has several inputs. The first is a set of focal nodes (e.g., cores of interest), selected interactively by the user. The second is the distances between nodes, designated $D(u,v)$, which correspond to the weights of edges in the graph. They are calculated according to the frequency of method calls and object moves between cores, as described in Section 4. The third input is the desired number of clusters. The output of the algorithm is a hierarchical clustering of the graph.

In the first step of the algorithm, the shortest distance between each node u and the closest focal node, $D^{focal}(u)$, is calculated. This is done using Dijkstra's algorithm on the focal nodes. Additionally, the maximum between the minimal distances is computed as

$$d_{max} = max_{v \in V}\{D^{focal}(v)\},$$

where V is the set of nodes in the graph.

In the second step, the distances, $D(u,v)$, between every pair of nodes u,v, are updated according to their proximity to focal nodes. This is done in order to create a fisheye-type effect in which nodes farther away from the focal points are displayed with less detail. As opposed to the regular fisheye technique, in which geometric distortion is used, our method moves the distortion to the clustering phase. We set the initial, joint average distance of nodes u and v and a focus node to

$$D^{focal}_{avg}(u,v) = \frac{D^{focal}(u) + D^{focal}(v)}{2}.$$

It should be noted that the focal node used in $D^{focal}(u)$ may be different from the one used in $D^{focal}(v)$. The distance $D(u,v)$ is distorted to form $D^{distorted}(u,v)$, the updated distance between nodes u and v, according to the following formula:

$$D^{distorted}(u,v) = \frac{D(u,v)}{1 + 3\frac{D^{focal}_{avg}(u,v)}{d_{max}}}.$$

The behavior of this formula, as a function of the fraction $\frac{D^{focal}_{avg}(u,v)}{d_{max}}$ is shown in Figure 7. As can been seen, the greater the average distance between the nodes and the closest focal node, the bigger the distortion. This behavior mimics the fisheye effect. Nodes in the periphery are less interesting and therefore have a higher probability of being clustered together, since they are perceived to be close.

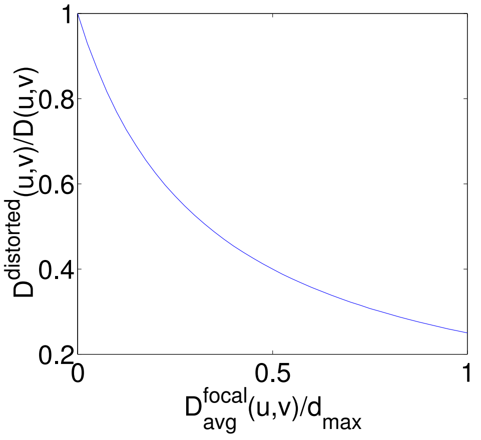


Figure 7: Distortion vs. distance from focus

Input: Distances between nodes; number of clusters
Algorithm:
1. Assign every node i to its own singleton cluster C_i.
2. Find the two closest clusters, designated C_i, C_j.
3. Merge C_i, C_j and update cluster distances.
4. Go to step 2 until the requested number of clusters is created.

Output: Clustering hierarchy of the nodes

Figure 8: Hierarchical clustering algorithm

In the last step of the algorithm the actual clustering is performed, using the distances computed in the previous steps. The algorithm is shown in Figure 8.

The distance between clusters C_i and C_j is calculated using a modified average distance metric. Only edges (distances) $e = (u,v) \in G$ directly connecting a node $u \in C_i$ and a node $v \in C_j$ (e.g., edges crossing the boundary between the clusters) are taken into account. The distance is

$$Dist(C_i,C_j) = \frac{\sum\limits_{(u,v)\in G, u\in C_i, v\in C_j} D^{distorted}(u,v)}{|\ \{e = (u,v) \in G | u \in C_i, v \in C_j\}\ |}$$

e.g., the sum of the lengths of the edges divided by the number of edges. This formula is a tradeoff between using the closest-neighbor distance that is fast, to calculating the exact average distance which gives better clustering results but is slower.

6 Implementation

We have implemented our system on top of *FarGo* [Holder et al. 1999a], a Java-based mobile object framework. FarGo contains extensive monitoring facilities [Gazit 2000] and uses a source to source compiler called *Fargoc* for generating proxies and other code used to implement support for mobile objects. Our implementation is Java based. We use the Java3D API for generating the visualization.

In each core (machine), a special *local profiling object*, used to collect events, is instantiated when the core is started. This object listens both to events generated by the Fargo monitor and to events generated by our modified Fargoc compiler. The local profiler object implements the active object design pattern [Lavender and Schmidt 1996] in order to be able to handle high event throughput without delaying the executing application.

The events generated by each core are forwarded to a main *event collection object*. If a hierarchical profiling architecture, described below, is used, the events arriving at the main event collector are only the subset of the generated events that are of interest to the user. They are forwarded to the event synchronization unit, described Section 6.2. Events may be stored to a file for offline visualization.

The visualization component receives the synchronized events, from which a consistent run of the application can be constructed. Events generated by the user, such as requests for re-clustering or zoom in / zoom out operations are fused together with the events collected from the system, in order to form a unified event queue that is visualized.

Below we describe how events are generated and synchronized.

6.1 Event Generation

One of the goals of a program visualization system is to generate events with minimal effort by the programmer and the user of the application being visualized, while perturbing the running application as little as possible. In this section we describe how this is achieved.

The interesting events are related to communication between mobile objects and movement of objects between cores. Since location transparency needs to be maintained when communication is performed between mobile objects, some kind of proxy needs to be used in order to forward the method call to the actual destination object. This proxy is generated either statically [Holder et al. 1999a] or dynamically [Acharya et al. 1996; Voy 1997]. This is where the event generation code is (automatically) inserted.

We assume the existence of a unique ID for each object in the system. Further, we assume the capability of adding code at the entrance and exit points of calls to the interfaces of mobile objects. These are the points to which the proxies interface, in order to send parameters and receive return values to/from remote objects.

In order to trace calls, when the execution flow enters an interface method of a mobile object, the ID of that object is recovered. The problem is finding out the ID of the caller object. In order to do that, we maintain a stack of object IDs through which the current thread of execution has passed: when entering an interface method of a mobile object, the current object ID is pushed onto the stack. When performing a return from such a method, the object ID is popped from the stack. It should be noted that in many cases mobile objects contain internal objects and these internal objects may make calls to other mobile objects. When a call is made from an internal object, detecting the ID of the caller object without using our ID stack method is not trivial.

Since objects can move between cores, the middleware should forward method invocations to the core where the destination object currently resides. Our system piggybacks essential information – the IDs of the callee and caller object – to the remote method invocation. Our method is able to handle exceptions by comparing the stack of called method names to the list of stored IDs and class types of objects on the object ID stack. When creating a new thread, the current object ID stack is copied to the child thread using thread-specific data. Bootstrapping the stack is done when entering the first method of a mobile object.

Generating events for movement of objects between cores is implemented by piggybacking onto the migration code supplied by the middleware. The required information (object ID, source and destination addresses) is easily obtainable.

Other types of actions for which events need to be generated include the creation and destruction of mobile objects and cores (e.g., connecting/disconnecting from the application network). This can be handled by adding code to the mobile object middleware through which these actions are performed or by tapping into an existing profiling interface of the middleware. We use the built-in monitor for generating these events.

6.2 Event Synchronization

One of the main challenges in visualizing distributed environments is the accurate depiction of events. Since in asynchronous distributed systems there is no way of knowing the real ordering of events, it is necessary to generate a visualization that is *consistent* with the events.

We base our solution to event synchronization on [Moses et al. 2004], where consistency of distributed environments with static objects was addressed, and extend it to support mobile object frameworks. In [Moses et al. 2004], the following is assumed:

1. There is a fixed (known) number of processes.
2. A process can perform two types of actions: sending a message to a different process and an internal computation, possibly modifying the process's local state. Receiving a message is considered an internal action.
3. The communication network and processes are reliable.
4. Messages sent by a single process to another process arrive in the order they were sent.
5. The network is asynchronous - there is no universal clock.

Since the visualization process is part of the distributed environment, it cannot know the relative order of actions performed by different processes. A way to solve this difficulty is to introduce *semantic causality*.

Definition 6.1 *With respect to a given algorithm run r, we say that an event e in r* semantically causes *e', denoted by $e \rightarrow e'$, if one of the following holds:*

1. *e and e' are on the same process, e occurs before e' and* indep(e,e') *does not hold (they are not semantically independent).*
2. *e and e' are on two different processes connected by a communication channel, e is a* send *event and e' is the corresponding* receive *event.*
3. *There is an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.*

Semantic causality requires information to be supplied by the user via the binary relation indep(a,b).

Let e and e' be two events of the algorithm. Let $An(e)$ and $An(e')$ be the animation segments of these events, respectively. We say that an animation $An(e)$ precedes an animation $An(e')$, denoted by $An(e) \prec An(e')$, if $An(e)$ completes before $An(e')$ starts.

Theorem 6.1 *An animation is consistent with the execution of the algorithm if and only if for every two algorithm events e and e', such that $e \rightarrow e'$ also $An(e) \prec An(e')$.*

That is, in order to ensure that the animation is consistent with the execution of the algorithm, we have to ensure that for every two events e and e', if $e \rightarrow e'$ then $An(e) \prec An(e')$.

A possible implementation of this requirement is called *receive synchronization*. In this method, reports of send and receive events are sent to the animation system immediately after they take place and there is no delay in the execution of the algorithm. The animation of the receive event is delayed until the corresponding send event has been animated.

We now turn our attention to mobile object environments. The main differences between this model and the distributed environments model, in the context of consistency, are:

1. Assumption 1 is violated. Both cores and objects might join or leave the network.
2. Objects might move between cores, which is not the case in classical distributed environments.
3. Assumption 4 is violated. Since objects might move, messages sent by a single object to another might be received out of order.

The first problem is addressed as follows. Dynamic creation and deletion of cores and objects are modeled as internal messages. A core / object is introduced to the animation system after its internal create event is received. A core / object is deleted from the animation system once a *deletion event* is received and all proceeding events have been animated.

To solve the second problem, object movement between locations is modeled as a method call between the sending and receiving cores. The parameters passed include the state and behavior (code) of the object that is being moved from one core to the other.

The third problem, out-of-order messages, should be solved by the middleware or the application. It is not a visualization problem, but rather an inherent problem. When this is solved, all that remains is to solve possible out-of-order messages to the visualization system. This can be solved by adding an event counter to each object and using the *receive synchronization* technique described above for visualization.

One possible approach to the event synchronization problem is to treat every mobile object as a separate process and perform synchronization accordingly. This means that all method calls into and out of every mobile object need to be synchronized. This solution is very fine-grained, since many method calls need to be traced. Also, since mobile objects may contain multiple threads of execution (that may cross from one object to the other), it is necessary to establish some kind of dependency relationship between the actions performed in different threads. Detecting such dependencies automatically may require extensive profiling of the execution of the application, which might slow it down by orders of magnitude. Our conclusion therefore is that using such a fine-grained synchronization method is not applicable in our case.

A second approach is to use a more coarse-grained view of the execution, performing synchronization at the core level. Much like regular distributed applications, each core is viewed as a separate process, and events notifying about communication between separate cores and activities internal to each core are emitted and synchronized. The internal events in each core are serialized. This may add redundant dependencies between activities that are independent in a core but is guaranteed to create a consistent visualization. The alternative of asking the user to explicitly define dependencies, is not viable in the context of our problem.

Messages sent between cores are modeled as messages sent between processes. The dependency between receiving the parameters for a message call and forwarding the parameters to the next core on the way to the destination core is handled automatically since these are two events that occur at the same core, one after the other. This is also true for messages sending the return value back to the caller core.

This scheme can be extended further to support scalability by using a hierarchy of synchronization units, constructed in a manner similar to what is discussed in Section 5. Each level in the hierarchy contains a synchronization unit. Events are forwarded to the next (higher) level only if they are not contained in the current level in the hierarchy.

Events showing average information that is periodically updated are not synchronized. For example, in our system events notifying the amount of communication between cores are periodically fired, yet not synchronized.

7 Case Study

Our system has been used for visualizing several applications, including a mobile object simulator, an e-commerce application [Joseph et al. 2000] and a distributed e-mail system (abbreviated DEM) [Bercovici 2004]. Due to the lack of space we focus on DEM.

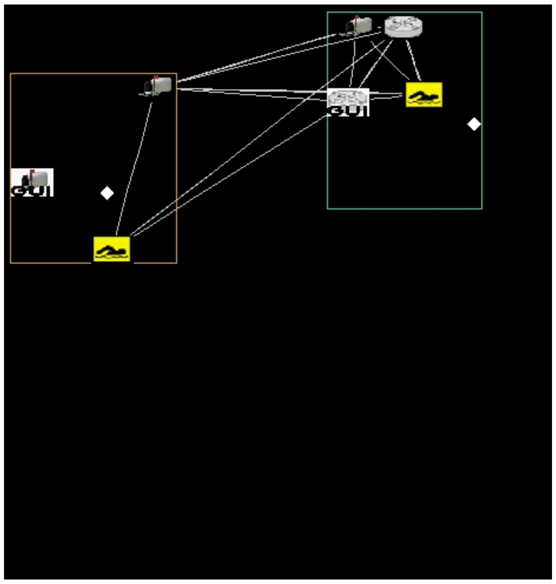
E-mail is one of the most popular Internet applications. Nowadays, e-mail architectures are governed by a server-centric design, which implies a handful of weaknesses such as a single point of failure, storage and processing stress, bottlenecks and inefficiency.

The goal of the DEM system is to overcome these drawbacks. The service is provided through the use of the participants' resources. Lightweight servers and users' mailboxes all scatter between participants' computers instead of residing on a single server (or cluster). Through the use of the mobile objects paradigm, the mailboxes and servers are able to travel on the "live" network, so that they continue their operation despite the fact that participants constantly join and leave the network. Most of the communication is done directly between users, thus removing the bottlenecks caused by mail servers. The system's components are replicated across numerous hosts, eliminating single point of failure problems. Storage and processing stress is reduced as participants take an even share of the burden. All of this yields a reliable and scalable system, with negligible operational and maintenance cost.

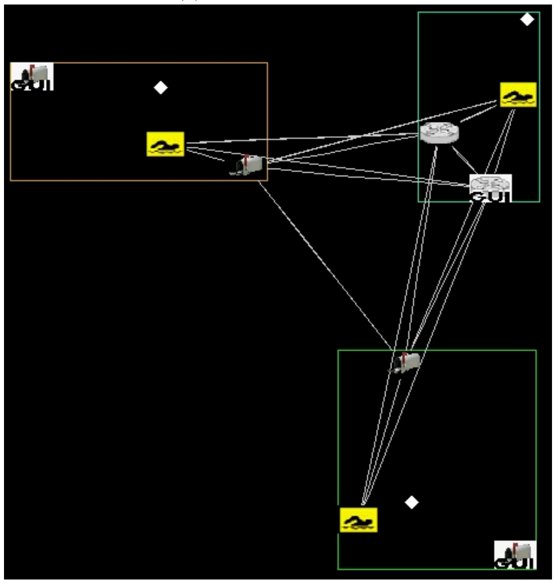
Visualization has been used during the development of this application – for debugging purposes as well as for managing and monitoring its deployment across the network. Due to the complexity of the architecture, its developer expressed a need for visualization at the very early stages of implementation. Using visualization, several problems were quickly discovered. For example, a case where an object does not flee from a core that is shutting down was uncovered.

In this application, icons have been used instead of the default rectangles, to represent the objects. The mailboxes are displayed using a mailbox icon. Servers are represented as gray disks. Yellow pools represent mailbox placeholders. Finally, the GUI is represented by a mailbox icon with a white background.

Figure 9 shows a visualization of the movement of a mailbox between computers. In Figure 9(a) there is one mailbox in each core. In Figure 9(b) a mailbox moved to a new core that connected to the service, shown at the bottom.



(a) Before movement



(b) A new core was created and a mailbox migrated to it

Figure 9: Mailbox mobility in the DEM system

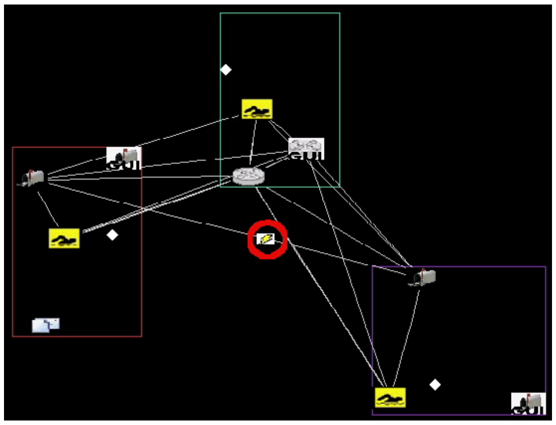


Figure 10: Sending an e-mail in the DEM system

Filtering of method calls was used in order to show specific interesting events. For example, Figure 10 shows an e-mail message being sent from the source mailbox directly to the destination mailbox. The message, in transit, is drawn inside a red circle. An accompanying movie can be found at http://www.ee.technion.ac.il/∼ayellet/Movies/FrishmanTal.mov.

8 Conclusion

We have presented a system for visualizing mobile object frameworks. The key features of these frameworks – object mobility, location transparency and distributed operation – are addressed by our system. A clustered graph is used to concurrently show the physical connections between cores as well as the logical connections between objects. A clustering algorithm, which is influenced by the areas of interest to the user, is used to provide a hierarchical, scalable context+focus view of the network. The overall complexity of the graph is user controlled. The visualization is dynamic: incremental graph layout and animation are used to depict changes in a smooth, comprehensible manner.

Our system has been used in several scenarios ranging from simulators to distributed e-mail and e-commerce applications. It has been used for monitoring, debugging as well as for presenting system architectures.

There are several avenues of future research. Additional levels of detail can be integrated into the visualization. The existing profiling infrastructure can be used to supply object-specific information such as memory usage and creation time. It has been implied that the delay between nodes in the network is of significance [Acharya et al. 1996; Gazit 2000]. This can be integrated into our framework. Finally, information about the cores themselves, such as thread count, memory usage and CPU usage can be integrated into the visualization.

Acknowledgements

This work was partially supported by European FP6 NoE grant 506766 (AIM@SHAPE), by the Israeli Ministry of Science, grant 01-01-01509 and by the Ollendorff foundation. Special thanks go to Sivan Bercovici for his work on the distributed e-mail system.

References

ACHARYA, A., RANGANATHAN, M., AND SALTZ, J. 1996. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, Springer-Verlag, J. Vitek and C. Tschudin, Eds., no. 1222 in Lecture Notes in Computer Science, LNCS, 111–130.

BERCOVICI, S. 2004. *Distributed Electronic Mail Project Report*. Available at `http://tochna.technion.ac.il/-project/DEM/html/index.html`.

BROWN, N., AND KINDEL, C. 1998. *Distributed Component Object Model Protocol — DCOM/1.0. Internet Draft*, January. Available at `http://www.microsoft.com/oledev/-olecom/draft-brown-dcom-v1-spec-02.txt`.

CARD, S. K., MACKINLAY, J. D., AND SHNEIDERMAN, B., Eds. 1999. *Readings in Information Visualization Using Vision to Think*. Morgan Kaufman.

DUDA, R. O., HART, P. E., AND STORK, D. G. 2000. *Pattern Classification*. Wiley–Interscience.

ELLSON, J., GANSNER, E. R., KOUTSOFIOS, L., NORTH, S. C., AND WOODHULL, G. 2002. Graphviz — open source graph drawing tools. In *Proc. 9th Int. Symp. Graph Drawing (GD 2001)*, Springer-Verlag, P. Mutzel, M. Jünger, and S. Leipert, Eds., no. 2265 in Lecture Notes in Computer Science, LNCS, 483–484.

ELMQVIST, N., AND TSIGAS, P. 2003. Growing squares: animated visualization of causal relations. In *Proceedings ACM 2003 Symposium on Software Visualization*, ACM, S. Diehl, J. T. Stasko, and S. N. Spencer, Eds., 17–26.

FRISHMAN, Y., AND TAL, A. 2004. Dynamic drawing of clustered graphs. In *Proceedings of the IEEE Symposium on Information Visualization, InfoVis*, IEEE Computer Society, M. Ward and T. Munzner, Eds., 191–198.

FURNAS, G. W. 1986. Generalized fisheye views. In *Human Factors in Computing Systems, CHI'86 Conference Proceedings*, Special Issue of ACM SIGCHI Bulletin, M. Mantei and P. Orbeton, Eds., ACM/SIGCHI, 16–23.

GAZIT, H. 2000. *Monitoring Support for Mobile Objects*. Master's thesis, Technion - Israel Institute of Technology.

HOLDER, O., BEN-SHAUL, I., AND GAZIT, H. 1999. Dynamic layout of distributed applications in fargo. In *Proceedings of the 1999 International Conference on Software Engineering*, IEEE Computer Society Press / ACM Press, 163–173.

HOLDER, O., BEN-SHAUL, I., AND GAZIT, H. 1999. System support for dynamic layout of distributed applications. In *19th International Conference on Distributed Computing Systems (19th ICDCS'99)*, IEEE, Austin, Texas.

JOSEPH, A., DAR, R., AND ALMOG, Y. 2000. *Active Market Project Report*. Available at `http://tochna.technion.-ac.il/project/amarket/html/home.htm`.

JUMPING BEANS, INC. 1999. *Jumping Beans White Paper*, May.

KAMADA, T., AND KAWAI, S. 1989. An algorithm for drawing general undirected graphs. *Information Processing Letters 31*, 1 (Apr.), 7–15.

KOHL, J. A., AND GEIST, G. A. 1996. The PVM 3.4 tracing facility and XPVM 1.1. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences (HICSS-29)*, IEEE Computer Society Press, H. El-Rewini and B. D. Shriver, Eds., vol. 1, 290–299.

KRAEMER, E., AND STASKO, J. 1993. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing 18*, 2, 105–117.

LAVENDER, R. G., AND SCHMIDT, D. C. 1996. Active object: An object behavioral pattern for concurrent programming. In *Pattern Languages of Program Design 2*, J. M. Vlissides, J. O. Coplien, and N. L. Kerth, Eds. Addison-Wesley.

MILOJICIC, D., DOUGLIS, F., AND WHEELER, R., Eds. 1999. *Mobility: Processes, Computers and Agents*. ACM Press.

MISUE, K., EADES, P., LAI, W., AND SUGIYAMA, K. 1995. Layout adjustment and the mental map. *Journal of Visual Languages and Computing 6*, 2, 183–210.

MOE, J., AND CARR, D. A. 2001. Understanding distributed systems via execution trace data. In *International Workshop on Program Comprehension*, IEEE Computer Society Press, 60–69.

MOSES, Y., POLUNSKY, Z., TAL, A., AND ULITSKY, L. 2004. Algorithm visualization for distributed environments. *Journal of Visual Languages and Computing 15*, 1, 97–123.

NORTH, S. C. 1995. Incremental layout in dynadag. In *Proc. 3rd Int. Symp. Graph Drawing (GD 1995)*, Springer-Verlag, F. J. Brandenburg, Ed., no. 1027 in Lecture Notes in Computer Science, LNCS, 409–418.

OBJECT MANAGEMENT GROUP. 1998. *The Common Object Request Broker: Architecture and Specification. Revision 2.2*, February.

PAUW, W. D., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J., AND YANG, J. 2001. Visualizing the execution of java programs. In *Proceedings of the International Seminar on Software Visualization*, Springer-Verlag, S. Diehl, Ed., no. 2269 in Lecture Notes in Computer Science, LNCS, 151–162.

REED, D. A., AYDT, R. A., NOE, R. J., ROTH, P. C., SHIELDS, K. A., SCHWARTZ, B. W., AND TAVERA, L. F. 1993. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of Scalable Parallel Libraries Conference*, IEEE Computer Society, 104–113.

STASKO, J. T., AND KRAEMER, E. 1993. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing 18*, 2, 258–264.

STASKO, J. T., DOMINQUE, J. B., BROWN, M. H., AND PRICE, B. A., Eds. 1998. *Software Visualization*. MIT Press.

SUN MICROSYSTEMS, INC. 1997. *Java Remote Method Invocation (RMI) Specification*, December.

TOLLIS, I. G., BATTISTA, G. D., EADES, P., AND TAMASSIA, R. 1999. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

TOPOL, B., STASKO, J. T., AND SUNDERAM, V. 1998. Pvanim: A tool for visualization in network computing environments. *Concurrency: Practice and Experience 10*, 14, 1197–1222.

OBJECTSPACE. 1997. *ObjectSpace Voyager Core Package: Technical Overview*, December.

WALSH, T., NIXON, P., AND DOBSON, S. 2000. Review of mobility systems. Tech. Rep. TCD-CS-2000-13, University of Dublin Trinity College, March.

WANG, Y., AND KUNZ, T. 2000. Visualizing mobile agent executions. In *Second International Workshop on Mobile Agents for Telecommunication Applications (MATA 2000)*, Springer-Verlag, E. Horlait, Ed., no. 1931 in Lecture Notes in Computer Science, LNCS, 103–114.