Algorithm Visualization For Distributed Environments [*]

Yoram Moses [†]
Dept. of Electrical Engineering
Technion

Zvi Polunsky
Dept. of Applied Math
The Weizmann Inst. of Science

Ayellet Tal[‡]
Dept. of Electrical Engineering
Technion

Leonid Ulitsky[§]
Dept. of Applied Math
The Weizmann Inst. of Science

Abstract

This paper investigates the visualization of distributed algorithms. We present a conceptual model and a system, VADE, that realizes this model. Since in asynchronous distributed systems there is no way of knowing (let alone, visualizing) the "real" execution, we show how to generate a visualization which is consistent with the execution of the distributed algorithm. We also present the design and implementation of our system. VADE is designed so that the algorithm runs on the server's machines while the visualization is executed on a web page on the client's machine. Programmers can write animations quickly and easily with the assistance of VADE's libraries.

1. Introduction

Algorithm visualization can assist in the design of algorithms, in the debug process, and while teaching algorithms to students and colleagues. When distribution is added to the environment, visualization can have an even greater value. Distributed algorithms are difficult to understand due to the added complexity of the interprocess communication and synchronization. Many activities occur concurrently at the various sites. Moreover, the activities depend on each other in many ways. Each state depends not only on the individual process, but also on the messages arriving from other processes.

Users can use a visualization in order to convey information about the way an algorithm might work. Programmers can use a visualization to debug their code; most standard debuggers do not support distributed programming in natural ways. Students can get a better insight into the algorithm and understand the way distribution is handled.

Visualization has been added to various phases of distributed computing [13]. It was added to parallel debuggers (e.g., [17], [18]), as well as to performance and monitoring tools (e.g., [12], [21], [11], [9], [10]). Algorithm animation systems aim to visualize higher–level abstract events than those of debuggers and monitors. This is true in the case of sequential algorithms (e.g., [7], [4], [25], [5], [27], [6], [1], [23]), as well as in the case of distributed and parallel algorithms (e.g., [24], [20], [26], [2], [14], [15]).

In asynchronous distributed systems there is no way to know what the "real" execution is. Each process in the system can only "remember" its own actions. It can also gain knowledge about actions performed by other processes through interprocess communication. It cannot, however, compute the relative timing of the actions performed by different processes. The visualization system, being itself part of the distributed asynchronous system, is no exception. The goal of an algorithm animation system for distributed environments is thus to produce a visualization which reflects as closely as possible the real execution of the algorithm.

As a result, a basic problem in visualizing distributed processes on a single display is that the display is constantly in danger of showing a picture that is locally consistent but globally inconsistent. The algorithm being visualized changes dynamically and the algorithm animation system needs to receive updates of the state from different sites. Since an immediate snapshot is impossible in an asynchronous distributed system ([16]), a "possible" snapshot should be constructed. This snapshot should be consistent with the execution of the distributed algorithm. It should present a possible state of the system consistent with the state in which the snapshot started and with the algorithm state in which the snapshot construction was completed.

There exist several algorithm visualization systems for

[*] Work has been supported by the Israeli Ministry of Science Grant 8544-3-97

[†] moses@ee.technion.ac.il
[‡] ayellet@ee.technion.ac.il, http://www.ee.technion.ac.il/∼ayellet
[§] ulitsky@cs.weizmann.ac.il, http://www.cs.weizmann.ac.il/∼ulitsky

parallel algorithms. For instance, in PARADE [26] the events of interest are ordered according to time–stamps. In the Animation Choreographer component of the PARADE system [14], it is also possible to manipulate the order of the display events by choosing a suitable ordering scheme. PAVANE [20] operates on the shared-memory tuple space architectures. In VISTOP [2], the "happened before" relation is utilized through either global breakpoints or traces. We propose to base the ordering of events upon the notion of *causal consistency*.

We present in this paper our conceptual model for an algorithm animation system for distributed algorithms. We also present a system, VADE (Visualization of Algorithms in Distributed Environments), which realizes the model. VADE is constructed so that the algorithm runs on the server's machines while the visualization is executed on a web page on the client's machine. End–users can thus watch the algorithm animation in a natural setting. The VADE architecture has several benefits. First, the algorithm code is protected. Second, the communication cost is very low, since only high–level operations are being sent over the network. Third, the framework provides a large degree of accessibility. Fourth, it allows overcoming Java applets' restrictions on communication capabilities, without sacrificing security. Finally, this architecture supports the enclosures of animations in online documents.

Programmers can write animations quickly and easily with the assistance of VADE's libraries, without having to be concerned either with the visual aspects of the animation or with the consistency maintenance.

The rest of the paper is organized as follows. In section 2 we discuss causal consistency of algorithm visualization and present our model. In section 3 we present the system's architecture. In section 4 we describe VADE from the end–user's perspective. In section 5 we describe VADE from the programmer's perspective. We conclude in section 6.

2. Visualization Consistency

A visualization system cannot be expected to represent the run exactly. We require, however, that it will be *consistent* with the run. In this section we define the formal notions of algorithm visualization consistency. We assume that: (1) The communication network is reliable. In other words, every message sent will eventually reach its destination. (2) Messages sent by a single process arrive in the same order they were sent. (3) The network is asynchronous. There is no universal clock.

Our goal is to produce a visualization that reflects the real execution as closely as possible. However, since the visualization process is just another process in the distributed environment, it cannot know the relative order of the execution of every two actions performed by different processes. A common way to work around this problem is to use the *potential causality* or *happened before* relation [16]:

Definition 2.1 *For two actions $a^{'}$ and $a^{''}$ we say that $a^{'}$ potentially causes $a^{''}$, denoted as $a^{'} \leadsto a^{''}$, if one of the following holds:*

1. *$a^{'}$ occurs before $a^{''}$ on the same process.*
2. *$a^{'}$ and $a^{''}$ are on two different processes connected by a communication line, $a^{'}$ is a* send message *action and $a^{''}$ is the corresponding* receive message.
3. *There exists an action b such that $a^{'} \leadsto b$ and $b \leadsto a^{''}$.*

The disadvantage of potential causality is that actions performed in a single process can be done in a different order without changing the semantics of the algorithm. In an algorithm animation setting, where the semantics of the algorithm is to be extracted, rather than the specific execution, it is useful to be able to present the algorithm in a different order. For instance, it is not necessary to postpone the animation of the actions performed after a *send* action until the latter is animated. Thus, we define below the *causality* relation.

Definition 2.2 *For two actions $a^{'}$ and $a^{''}$ we say that $a^{'}$ causes $a^{''}$, denoted as $a^{'} \rightarrow a^{''}$, if one of the following holds:*

1. *$a^{'}$ and $a^{''}$ are on the same process, $a^{'}$ occurs before $a^{''}$, and their order of execution cannot be changed without altering the algorithm semantics.*
2. *$a^{'}$ and $a^{''}$ are on two different processes connected by a communication line, $a^{'}$ is a* send message *action and $a^{''}$ is the corresponding* receive message.
3. *There exists an action b such that $a^{'} \rightarrow b$ and $b \rightarrow a^{''}$.*

The difference between this relation and the *potential causality* relation lies in the first condition. If $a^{'}$ occurs before $a^{''}$, it does not necessarily mean that $a^{'}$ causes $a^{''}$. While potential causality can be identified automatically, causality cannot. We thus need to assume an external source of knowledge regarding causality. It is the responsibility of the programmer to supply this information.

The process actions are modeled as events. The initial local state of process p is denoted by s_0^p. The local state s_t^p of process p at time t is modeled by a sequence of events that occur until time t. We will use a slightly different model than the usual one by allowing actions that take more than one time unit. This is more suitable for modeling animations, since an animation event might take more than one round. An action a is modeled by two events: a_e is the *execution* of the action, while a_c is its *completion*.

An execution of the algorithm by a process p is modeled by a sequence S^p of the process local states $S^p =< s_0^p, s_1^p \ldots >$. The modeling of the execution of the entire algorithm is done by arranging the set of actions of all participants on a global time scale, so that there is no contradiction with the causality relation between the actions.

The events occurring in the algorithm processes are reported to the algorithm animation system. The animation system can store the event reports and build a "model of the algorithm execution" E. This allows the system to build an animation based not only on the last event reported, but also on a set of previously received reports. The animation system need not always build an animation segment as a result of an event report. Sometimes the animation system saves them and animates them later. Sometimes the report serves as a tool for synchronizing the animation, and the event itself is not animated. A number of frame elements, triggered by different algorithm events can have the same frame number, reflecting the simultaneous execution of the algorithm by different processes.

Let S^p be an execution of the algorithm by a process p. We can build another possible execution of the algorithm S_1^p by rearranging the sequence of events such that it is consistent with respect to the $causality$ relation, i.e., in the resulting sequence: (1) For every action, its *execution* event must precede its *completion* event. (2) For every two actions a and b such that $a \to b$, a_c must precede b_e.

Note that this definition not only allows us to rearrange the events, but also allows us to model simultaneous execution of events, when other events are allowed to appear between the execution and completion events of an action. We require that the animation represents a sub–sequence of a member in the set of consistent runs of the algorithm.

Definition 2.3 *Let r be a run representing the real execution of the algorithm. Let $R(r)$ be the set of all consistent runs. Let $F =< F_0, F_1, \ldots, F_t >$ be an animation of this run. Let $E =< E_0, E_1, \ldots, E_t >$ be the sequence of models used to build the animation frames F. The animation F is said to be* consistent *with the algorithm run r if and only if the sequence E is identical to a sub-sequence of global states of a run $r^{'}$ that is a member of the set of runs $R(r)$ produces by r.*

It is left to show how to achieve visualization consistency. Let a and b be two interesting events of the algorithm. Let $An(a)$ and $An(b)$ be the animation segments of a and b, respectively. We say that an animation $An(a)$ precedes an animation $An(b)$, denoted as $An(a) \prec An(b)$, if $An(a)$ completes before $An(b)$ starts. We claim the following:

Theorem 2.4 *An animation is consistent with the execution of the algorithm if and only if for every two algorithm events a and b, such that $a \to b$ also $An(a) \prec An(b)$.*

Thus, to ensure visualization consistency we have to ensure that for every two algorithm events a and b, if $a \to b$ then $An(a) \prec An(b)$. This requirement is a special case of the requirement for *casual ordering* of events [3]. The casual ordering is respected if the following condition holds: If $send(m_1, p_1, p_3) \to send(m_2, p_2, p_3)$ then $receive(m_1, p_3, p_1) \to receive(m_2, p_3, p_2)$.

In the algorithm animation context, the display of $An(a)$ is analogous to the visualization system's "receiving" a. The causality relation between two such receive operations is equivalent to the $precedes$ relation between the animation segments. Moreover, we can regard the execution of an event in the algorithm as a "send" of the event to the visualization system. Thus, the visualization ordering requirements are transformed into the causal event ordering requirements.

2.1. Implementation of casual ordering

To implement the visualization ordering requirement, it is enough to implement it for pairs related by immediate causality. The transitivity guarantees that it will hold for the general causality case. This claim is true if all the events are reported to the visualization system. Note, however, that not all the events need to be reported, but rather only those that need to be animated and all the $send$ and $receive$ actions.

In VADE, when two events occur in the same process, and the events are reported to the algorithm animation system, the first event is animated before the second by default. If one event is a $send$ and the other is the corresponding $receive$, we ensure the animation ordering requirement in one of two ways: *send synchronization* and *receive synchronization*. Unlike some other systems (e.g, [3] [19] [22]), we need not add any additional information regarding the history. We elaborate below.

Send Synchronization: In this scheme, the report on the $send$ is sent to the visualization system before the actual action takes place. Then the process waits for a confirmation from the algorithm animation system. Only when the confirmation arrives, the actual $send$ can be performed. The report on the $receive$ event is sent to the animation system after the actual message is received, thus ensuring that the animation of the *receive* event is performed after the animation of the *send* event.

Lemma 2.5 *In the Send Synchronization scheme, $An(send) \prec An(receive)$.*

The disadvantage of the method is that the actual execution of the algorithm is altered by the fact that it is being visualized. This might be a problem when visualization is used for monitoring and debugging. However, when the system is used merely as an aid for explaining an algorithm, *send synchronization* is a satisfactory solution.

Receive Synchronization: In this scheme, the actual synchronization of the animation with the algorithm events is performed in the animation system. The reports of the $send$ and $receive$ events are sent to the animation system immediately after the actions take place and there is no delay in the execution of the algorithm. However, the animation of the $receive$ event is delayed in the animation system until the corresponding $send$ event has been animated. To support this scheme, the animation system needs to provide tools for suspending the execution of certain events until other events occurred.

The scheme is implemented as follows. Two counters are maintained for each communication channel – one for the send actions, and the other for the receive actions. When a *send* report arrives, it is animated immediately. When the animation is completed, the counter of the send actions is incremented. When a *receive* report arrives, the animation system checks the counters of both the send actions and the receive actions for a given channel. The animation starts only if the number of send actions is larger than the number of the receive actions. Otherwise, the animation is delayed.

Lemma 2.6 *In the Receive Synchronization scheme,* $An(send) \prec An(receive)$.

The major advantage of the *receive synchronization* scheme is that the execution of the algorithm is not being changed. The disadvantage of the method is that it may require queuing many reports.

3. System Architecture

There exist various models for sequential algorithm visualization systems over the Internet [1]. In the *X model* the visualization program runs on the remote machine and interacts with the X server on the local user's machine. The *Java model* allows the execution of the animation on the user's machine after the code has been transferred. The *Mocha model* [1] bridges between the two models by exporting the interface code, while executing the algorithm on a server. We follow the latter approach, and extend it to support distributed computing.

The architecture of VADE is illustrated in Figure 1. The algorithm is executed on the server's machines, while the animation and the GUI are executed on the client's machine. The client–side processes run at a WWW browser. In Figure 1, the '$\rightarrow$' relation indicates communication, and the '$\Rightarrow$' relation indicated forking the processes as threads.

The various processes are written in Java. The processes on the server side are Java applications, while the processes on the client side are Java applets. Their code is down–loaded by the WWW browser over the Internet, compiled and run within the browser. This allows Internet users to watch the animation in their browsers. The communication between the server processes and the client processes are performed with the TCP/IP protocol.

When the end–user opens a web page on the client side, the *main client* process threads a GUI process, which is in charge of animating the algorithm. The *main client* informs the *main server* of the type of the algorithm to be animated. Upon receiving a reply from the server, and according to

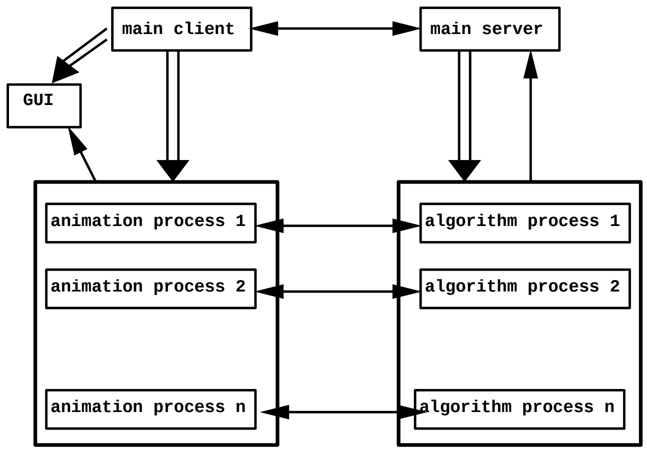


Figure 1. The VADE architecture

the data received, it threads the required number of animation processes. This number matches the number of algorithm processes. The GUI process gets requests from the animation processes to animate the interesting events of the algorithm, and modifies the screen accordingly.

All the *animation processes* run on the same host on the client side, as concurrent threads. As a result, scheduling is performed by the operation system, and no special handling is needed by the visualization system. This is different from the *algorithm processes* which run on different machines.

The *main server* is in charge of executing the algorithm. Upon receiving a request from the client, which includes the type of the algorithm to execute, the main server threads the processes which are necessary for executing the algorithm. In order to determine the nodes on which the algorithm should run, the main server maintains a table of the available hosts on the server side. It also maintains a dynamic table containing information regarding the processes and their hosts. When an algorithm process terminates, it informs the server, and the process table is being updated.

The task of the *algorithm processes* is to execute the algorithm. Before an algorithm process starts running the algorithm, it needs to establish a connection with the corresponding *animation process*. During execution, upon reaching an interesting event, it sends a message, containing the relevant information regarding the event, to its related animation process. Interesting events are either events of the algorithm, or communication events (i.e., *send* and *receive*). Both synchronous communication and asynchronous communication are supported. Recall that if all the *send* and *receive* events are reported to the animation processes, the animation consistency is automatically maintained by VADE.

The above architecture has a few benefits. First, the algorithm code of the provider is protected. It is only the animation code that is being down–loaded by the client. Second, the communication cost is very low, since only high–

level operations are being sent over the network, rather than the detailed frame description of the animation. Third, the framework provides a large degree of accessibility. The client need not have the resources needed for running the distribute algorithm. Fourth, this scheme overcomes Java applets' restrictions on communication capabilities. Java applets, for security reasons, avoid the creation of new processes on other machines, as well as the ability to update files. These, however, are necessities in distributed environments. Finally, utilizing the web makes it possible to support enclosures of animations in online documents – an invaluable tool for illustrating the functioning of algorithms.

4. End–User Perspective

In order to run an algorithm and watch its animation, the end–user should open a web page and select a specific algorithm. Each algorithm has its own dedicated page. An algorithm web page displays a view, or multiple views, of the algorithm animation, along with a couple of panels that support the interaction with the animation, as illustrated in Figure 2. (See also the color plate.)

A *control panel* enables the user to play the animation, pause it at any point during its execution, and resume its execution. The user can also choose a node which starts the execution, by pressing the *Starting node* button.

A *configuration panel* makes it possible to build new configurations of the network by adding or removing nodes and communication channels. With the *Select configuration* button the user can either select a *default configuration*, in which case the system generates a configuration, or *build configuration*, which allows the user to build a network configuration using the *Add node* and *Add edge* buttons.

Figure 2 displays snapshots from an animation of the snapshot algorithm [8]. This algorithm builds a snapshot of the network. Since an immediate snapshot is impossible, a "possible", consistent, snapshot is constructed. In our example, the nodes randomly exchange data. Any node can start the snapshot algorithm either randomly or after receiving a *marker* from a neighbor. From this point on, the node proceeds with its standard message exchanging algorithm while saving the values of all messages arriving. The messages on each communication channel are saved until a *marker* is received on that channel. The node completes the snapshot when markers are received on all the channels.

In our example, the animation consists of three views. The upper view displays the detailed execution of the algorithm. The nodes are displayed as squares with their values in the center. The colors encode the state of the node. A node is colored red if it does not participate in the snapshot algorithm. Its color is changed to blue after it received the first marker. Finally, the color of the node changes to green after that node completed the snapshot. At the end of the algorithm, all the snapshot nodes should be colored green. The communication channels are displayed as lines connecting the nodes. The messages are represented as circles, and the markers as arrows. Both the messages and the markers travel along the communication channels.

The middle view displays the state of snapshot, built by the algorithm. Before any node started the algorithm execution, this view is empty. When the node enters the snapshot algorithm, its value is added to this view. Similarly, when the node receives a message, the message is added to the snapshot and to this view.

The lower view shows the sum of the "money" in the snapshot. The left square displays the total sum in the system before the algorithm started to run. The right square displays the sum accumulated during the snapshot. When the snapshot is completed, the two numbers should be equal.

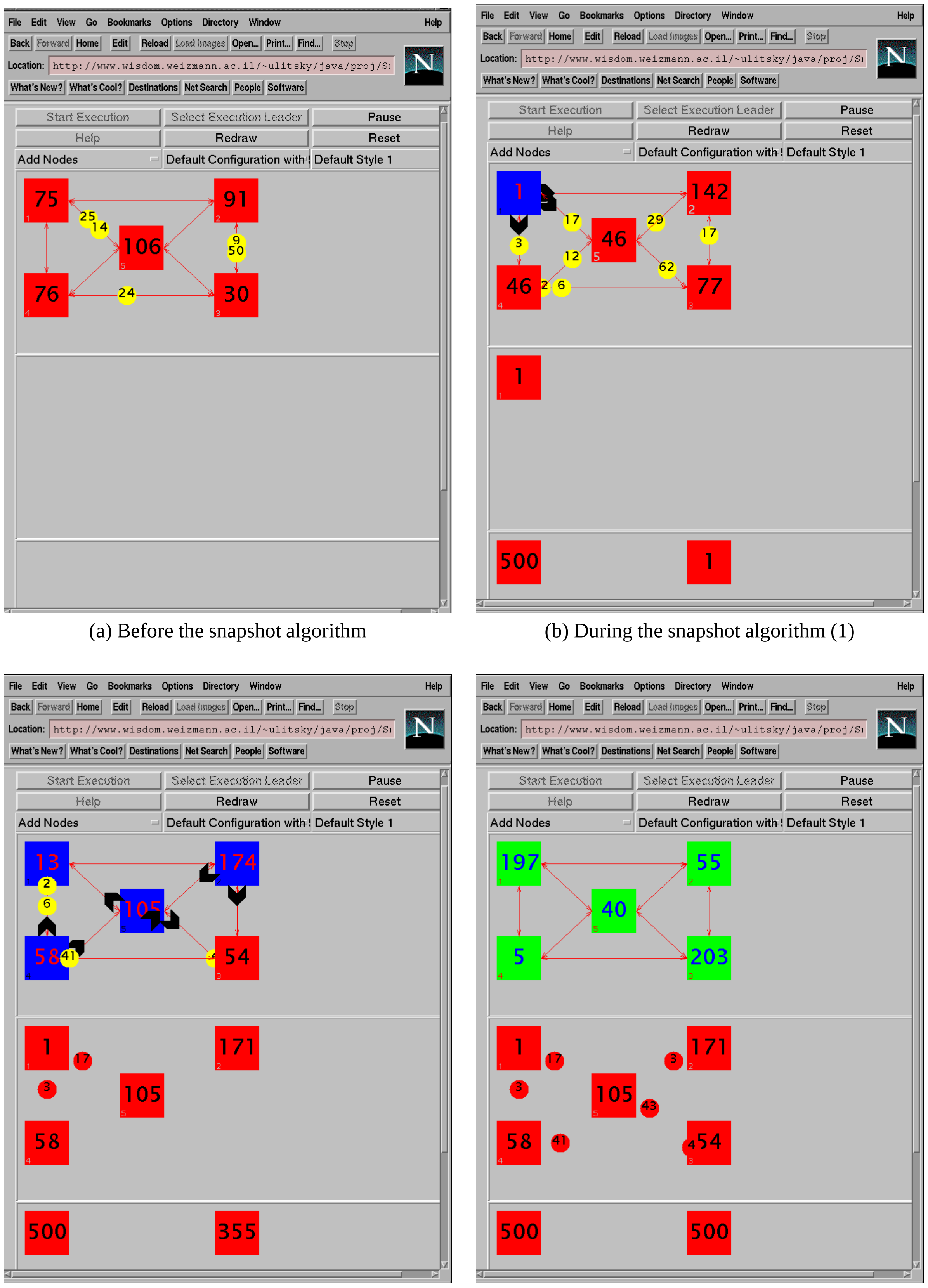
Figure 2(a) displays the state of the system before the snapshot algorithm has begun. Figures 2(b)–2(c) demonstrate the state of the system during the algorithm. Finally, Figure 2(d) illustrates the state of the system after the snapshot algorithm terminated.

5. Programmer Perspective

Following the event–driven approach [4], strategically important points of the algorithm are augmented with interesting events, for which animations should be produced. The task of animating a distributed algorithm is thus divided into four steps. First, the interesting events should be identified. Second, the algorithm should be implemented and annotated with the interesting events. Third, the code should be augmented with calls to the animation system's synchronization functions (i.e., *send* and *receive* events should be reported). Finally, the animation needs to be implemented.

In our framework, a large portion of the animation implementation is left for the animation system. Our system provides the GUI and a few libraries which facilitate the creation of animations, as described below. In addition, the programmer need not be concerned with maintaining the consistency of the animation, which is automatically done by the animation system, as discussed in Section 2. In our web setting, the programmer needs also create a web page that contains the views of the animation. VADE provides the basic web page which contains the control panel and the configuration panel, as described in Section 4. The page can be modified to accommodate any number of views, as well as adding algorithm–specific options to the control panel and to the configuration panel.

The animation of the events should be written in Java. Figure 3 summarizes the VADE class tree. The classes contained in the tree build up the animations, and their visualization is supported by VADE. Derived classes are at the right. They inherit the fields and the methods of the classes


File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Load Images Open... Print... Find... Stop
Location: http://www.wisdom.weizmann.ac.il/~ulitsky/java/proj/S
What's New? What's Cool? Destinations Net Search People Software
Start Execution
Select Execution Leader
Pause
Help
Redraw
Reset
Add Nodes
Default Configuration with
Default Style 1
75
91
106
76
30
25
14
9
50
24
1
142
46
77
17
29
12
62
3
2
6
500
13
174
105
58
54
41
171
355
197
55
40
5
203
43

(a) Before the snapshot algorithm

(b) During the snapshot algorithm (1)

(c) During the snapshot algorithm (2)

(d) The result of the snapshot algorithm

Figure 2. Snapshots from the visualization of the snapshot algorithm

they are derived from. The main class that describes the network is the *NetGraph*. It includes nodes (the *NetNode* class), communication channels (the *NetLink* class) and various types of tokens (the *NetToken* class) that can be exchanged by the nodes on the communication channels. The *NetToken* class includes messages (*NetMessage*), markers (*NetMarker*), or other types of tokens (*NetUserToken*). To accommodate other objects relevant for the animation of the algorithm, the user can use the class *NetUserObject*.

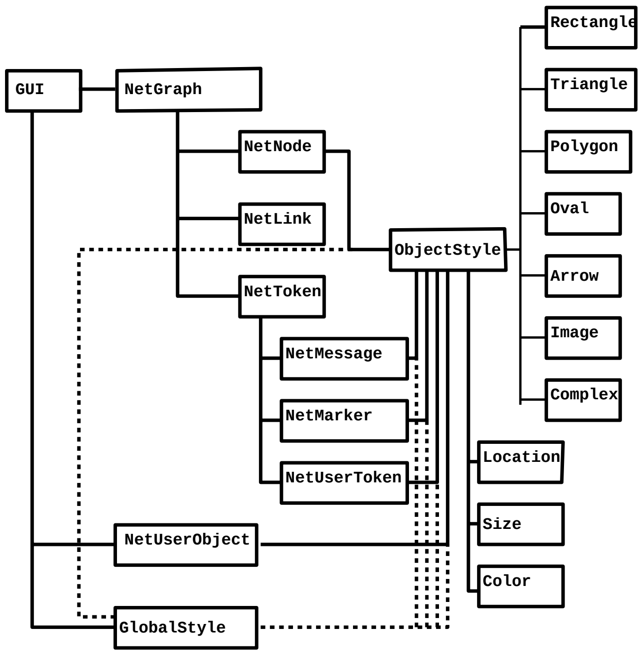


Figure 3. The structure of the animation classes

A major principle we follow is that **what** the animation presents is distinguished from **how** it does it. In other words, the animation contents are distinguished from visual appearance on the screen (its *style*). For instance, suppose that the user wishes to animate a “send” event. It can be visualized in various ways – nodes can be displayed in various colors and shapes, communication channels can be drawn in different line thicknesses, the messages can travel quickly or slowly etc. While *NetNode, NetLink*, and *NetToken* contain the information about the contents of the animation (i.e., a message X travels on a channel Y from a node A to a node B), the *style* of the animation should contain the information about the actual visualization. Default values for the style parameters are set by the system. Thus, the programmer can quickly generate a prototype animation by specifying only the information regarding the contents of the animation. At later stages, the programmer can experiment with various animations by modifying the *style* classes, without altering the network classes. Note the dashed lines in Figure 3 that mark the style information.

The style of the animation is stored in two classes: the *GlobalStyle* class and the *ObjectStyle* class. While the *GlobalStyle* class defines the global visual appearance of the animation, the *ObjectStyle* class defines the style of a specific object. The *GlobalStyle* class also specifies the global aspects of the animation which are not related to the objects, such as the background color and the number of frames used for animating each of the interesting events of the algorithm. Each of the possible shapes has its own *paint* method. Note that a node can be displayed, in addition to the simple geometric shapes, as an image, or as a complex shape which is built out of simpler shapes.

To better demonstrate the difference between *ObjectStyle* and *GlobalStyle*, suppose that the nodes were defined in the *GlobalStyle* as red rectangles. If the user changes the shape and the color in the *GlobalStyle* to a green triangle, all the nodes will be repainted as green triangles. However, if only a specific *ObjectStyle* is to be changed to a green triangle, that specific node will be repainted as a green triangle, while the rest of the nodes will remain red rectangles.

To summarize, VADE facilitates the task of creating a visualization by providing classes prevalent in distributed applications. It also supplies support for creating a web page for the animation. By distinguishing between the contents and the style of the animation, it becomes easy to quickly generate a prototype animation, which is often sufficient (for instance, for debugging purposes). Should a fancier animation be desired, it can be easily done by modifying the style classes, without altering the contents of the animation. Finally, VADE frees the programmer from having to be concerned with maintaining the consistency of the visualization with the algorithm executed.

6. Conclusion

Distributed algorithms can be very difficult to grasp, and hard to implement and debug. Visualization can assist in all these tasks. We presented in this paper a system, VADE, and a conceptual model, for visualizing algorithms in a distributed environment.

With VADE, distributed algorithm animations can be produced quickly and easily. The system provides libraries that facilitate the generation of visualizations. It clearly distinguishes between the contents of the animation and its visualization attributes. This allows the user to experiment with various animations for the same running algorithm. Moreover, VADE automatically maintains the consistency of the picture presented with the algorithm executed.

End–users can view the animation in the current natural environment – the web. A control panel enables the users to control the execution of the animation, while a configuration panel allows the users to interact with the system by constructing various distributed configurations.

VADE is designed so that the algorithm is executed on the server's machines, while the animation and the GUI are executed on the client's machine, as Java applets. This architecture allows a large degree of accessibility, code protection, and a low communication load.

References

[1] J.E. Baker, I.F. Cruz and G. Liotta. Algorithm animation over the World–Wide Web. , 1996.

[2] T. Bemmerl and P. Braun. Visualization of message passing parallel programs with the TOPSYS parallel programming environment. *Journal of Parallel and Distributed Computing*, 18:118–128, 1993.

[3] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[4] M.H. Brown. *Algorithm animation*. MIT Press, 1988.

[5] M.H. Brown. Zeus: a system for algorithm animation and multi-view editing. *Computer Graphics*, 18(3):177–186, May 1992.

[6] M.H. Brown and M.A. Najork. Collaborative active textbooks: a web-based algorithm animation system for an electronic classroom. *SRC Report 142*, 1996.

[7] M.H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, 1985.

[8] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[9] M.T. Heath, A.D. Malony and D.R. Rover. The visual display of parallel performance data. *IEEE Computer*, 28(11), 21–28, 1995.

[10] M.T. Heath, A.D. Malony and D.R. Rover. Parallel performance visualization: From practice to theory. *IEEE Parallel Distrib. Tech.*, 3(4), 44–60, 1995

[11] H. Jakiela. Performance visualization of a distributed system: A case study. *IEEE Computer*, 28(11), 30–36, 1995.

[12] J. Joyce, G. Lomow, K. Slind and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, 1987.

[13] E. Kraemer and J.T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[14] E. Kraemer and J.T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations *Proceedings of the 8th International Parallel Processing Symposium*, 902–908, 1994.

[15] E. Kraemer and J.T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1), 36–46, 1998.

[16] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558-565, 1978.

[17] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[18] C.M. Pancake and S. Utter. Models for visualization in parallel debuggers. *Proceedings of Supercomputing '89*, 627–636, 1989.

[19] M. Raynal, A, Schiper and S. Toueg. The casual ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

[20] G.-C. Roman, K.C. Cox, D. Wilcox and J.Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *J. Visual Languages Comput.*, 3(2): 161–193, 1992.

[21] S.R. Sarukkai. Performance visualization and prediction of parallel supercomputer programs: An intern report. *Tech. Report 318, Indiana University* , 1990.

[22] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement casual ordering. *Third International Workshop on Distributed Algorithm, Lecture Notes in Computer Science 393*, 219–232, 1989.

[23] M. Shneerson and A. Tal. Visualization of geometric algorithms in an electronic classroom. *Visualization '97*, 455–457,576, 1997.

[24] D. Socha, M.L. Baily and D. Notkin Voyeur: Graphical views of parallel programs. *SIGPLAN Notices* 24(10: 206–215, 1989

[25] J. Stasko. Tango: a framework and system for algorithm animation. *IEEE Computer*, September 1990.

[26] J. Stasko and E. Kraemer A methodology for building application-specific visualizations of parallel programs. *J. Parallel Dist. Comut.* 18)(1):258–264, 1993.

[27] A. Tal and D.P. Dobkin. Visualization of geometric algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1:194–204, 1995.