Interactive Visual Exploration of Distributed Computations

Delbert Hart, Eileen Kraemer, and Gruia-Catalin Roman
Department of Computer Science
Washington University in Saint Louis

Abstract

Program understanding is central to the development of distributed computations, from the initial coding phase, through testing and debugging, to maintenance and support. Our goal is to support programmers unfamiliar with a particular distributed computation in developing a reasonable understanding of the workings of a program, without requiring that they examine the details of the code itself. Toward this goal, we propose query-based visualization, a novel exploratory approach to understanding distributed computations. The key features of the approach are the use of queries as a device for searching the state space, visual presentation techniques adapted from program animation, and the ability to navigate through the state space using visual interactions. All views correspond to globally consistent snapshots of the computation. A working prototype demonstrates the technical feasibility of the approach.

1. Introduction

Distributed computing presents developers with unprecedented challenges and the search is on for powerful new tools to assist designers and programmers. The range of tools under consideration includes new languages, application frameworks, packages, visualization systems, and more. Regardless of the direction one might pursue, program understanding is central to all development activities. In this paper, we take a somewhat extreme perspective on program understanding by posing the question whether it might be possible for a programmer unfamiliar with a particular distributed computation to develop a reasonable understanding of its workings without actually examining the code. We believe that this is the kind of mindset one needs to adopt if one seeks to develop fundamentally new strategies for program understanding. Having taken this position, we are forced to view the execution of a distributed computation as an object of empirical study subject to live exploration.

Our strategy treats a distributed computation as a database containing the state of the individual processes executing across a network. The state of each process changes due to local computations, as a result of message passing activities, and as a consequence of process creation and termination. Queries provide the mechanism by which exploration is conducted. Navigating through the state space of the computation entails the gradual refinement of the queries posed by the programmer.

In order to avoid the kind of misleading information that can result from the presence of communication delays, all queries are evaluated logically with respect to consistent global states of the executing system. Typical queries are viewed as persistent and are evaluated after each change in the system state. Efficient snapshot algorithms are central to the success of this approach. The ability to minimize the number of processes required to participate in the construction of the snapshots helps to minimize perturbation.

Finally, there is the issue of how to present the potentially large volume of information extracted by the queries, especially when the query evaluation amounts to a continuous monitoring of selected properties of the system execution. The solution we employ is borrowed from our past successful experience with program visualization. Information is presented as on-line three-dimensional animations of the evolving system state. The combination of queries and visualization give the method its name, query-based visualization. Without loss of generality, our initial implementation employs the PVM [12] library and relies on certain assumptions about the way the computation is structured in order to achieve efficient query processing and snapshot collection.

Figure 1 shows the logical organization for a system that implements the query-based visualization paradigm. Each user starts an exploratory session by establishing a communication path, indicated by the gray arrows, to an executing distributed computation. A monitoring package embedded in the code of the distributed computation provides information about the state of the individual processes to all interested parties–multiple user sessions may connect to a given computation. A snapshot manager, responsible for snapshot construction and query processing, uses this information to construct a consistent global snapshot of the computation and passes it to an animation program which generates a visual representation to be examined by the user. Interactive controls allow the user to navigate through the visualization and to construct
imagery which, through a decoding process, is assigned query semantics and is made available to the snapshot manager. The latter informs the monitoring package about what information is needed to construct the next series of visualizations. Multiple users may be involved in examining the same computation, each one making different independent requests and viewing continuously updated smooth animations of selected aspects of the system’s global state.

In the following sections, we describe a computational model for distributed programs, algorithms to construct consistent snapshots of the computation, and methods of interaction with the snapshot collection and visualization systems. Section 2 provides a characterization of the distributed computing model assumed in this paper. Section 3 discusses two algorithms for generating continuous snapshots, one for snapshots of the entire computation and the other involving only the monitored processes and selected state attributes. Section 4 explains how visualizations are generated from the consistent snapshots and illustrates the exploratory process. Section 5 discusses related work. Finally, Section 6 provides a summary of our preliminary results.

2. Computational Model

Although the ultimate goal is to permit exploration of arbitrary distributed computations, our current work focuses on a restricted class whose communication patterns are sufficiently structured so as to facilitate a reasonably efficient implementation of the query-based visualization model. The key feature we exploit is the fact that the overall computation can be abstracted to an interleaving of atomic state changes involving one or more processes—by analogy with databases we call such state transitions transactions. Transaction processing applications are a natural choice for obtaining global state information, since their structure matches the logical actions performed by the application. A money transfer, for example, may involve two processes located at different points of the network and it is highly desirable to be able to treat the money exchange as an atomic operation on the state of the two bank accounts. Many multi-phased computations also fall in this category of applications whose structure reflects the logical computation. The N-body problem, for instance, employs alternating communication and computation phases in which information about the state of neighboring regions of space is exchanged in order to compute the forces acting on each body and its next position in space. In some cases, the transaction concept can be superimposed on computations which otherwise execute in a highly unstructured manner.

Underlying this view of distributed computing is the reality of message-based communication via reliable FIFO channels with the added complication that processes may be created dynamically and may terminate at any time. By and large, the application code need not be written in any special way in order for us to visualize it. The only exception is the need to specify the end of each transaction in each of the participating processes.

2.1 Transactions

A distributed computation consists of a set of processes that work together to achieve a common goal. Each process exports a set of attributes that reflect the state of the process. The process’s state changes when an event occurs at the process. The event sequence is a history recording the changes undergone by the process. An event \( x \) is characterized by:

- \( \pi.x \): the process with which this event is associated
- \( \sigma.x \): the state of the process \( \pi.x \) immediately after \( x \) occurs
- \( \delta.x \): a positive number that reflects its position in the process’s sequence of events
- \( \tau.x \): an event type indicating the nature of the event, i.e., \( \text{send, receive, mark, and local} \)

A local event represents a state transition within a single process. A mark event indicates that the process has completed its participation in the current transaction. A matching send/receive event pair is
called a communication. In addition, the events init, start and stop (special instances of the events send, receive and mark) denote a request to create a process, the start of a process, and the termination of a process, respectively.

Formally, we define a distributed computation as a triple \((\Sigma, <, \sim)\) where \(\Sigma\) represents a set of events, \(<\) is a partial order (call it a computation graph) corresponding to the happened-before relation, and \(\sim\) is an equivalence relation capturing the notion that two events are part of the same transaction.

Informally a distributed computation is well formed (i.e., an appropriate model) if a) for each process there is a total ordering of events within that process, b) the only interaction between processes is through message passing, c) a send event is found in a process’s sequence of events if and only if there is a corresponding receive event in the sequence of events in some other process, d) transactions are equivalence classes over events, and e) the send and receive events of a communication belong to the same transaction. A more precise definition of a well formed computation is given in [3].

A well formed distributed computation involving four processes is illustrated in Figure 2. It can be seen that in this instance the transactions form a partial order consistent with the happened-before relation over events. This property is true in general of transactions. If we use \(trans(x)\) to denote a function which returns the transaction to which the event \(x\) belongs, the happened-before relation can be extended to transactions, i.e., a transaction \(\gamma_1\) happened-before a transaction \(\gamma_2\) if and only if there is an event in \(\gamma_1\) that occurred before some event in \(\gamma_2\). Transactions form the boundaries in the process executions that we will use to construct consistent global snapshots.

This computational model describes a class of distributed computations that are amenable to efficient monitoring. Transactions correspond to logical actions performed by the computation, they are of interest because we want to provide algorithm understanding without requiring the detailed knowledge necessary to understand the computation’s code.

### 3. Snapshot Construction

In this section we discuss two algorithms for obtaining on-line consistent global snapshots. Specifically, the algorithms construct snapshots of the computation that include the data requested by the queries that the user has issued. The snapshots are taken at the boundaries of the transactions of the application. The sequence of snapshots constructed provides a view of the application’s logical progression. The user is able to observe this progression via the visualization environment discussed in the next section. The first algorithm covers the special case in which the user wants to monitor all available data all of the time. Next, we provide a general algorithm that allows the queries to change and to selectively monitor parts of the application.

We assume a well formed computation augmented with a reporting mechanism that transmits information about transaction completion and the resulting state of a process to a snapshot manager. Communication with the snapshot manager is assumed to be message based, reliable, and FIFO. We attempt to limit any interference with the ongoing computation to be bound by the amount of monitoring done, to limit the amount of information transmitted to the snapshot manager, and to constrain the reporting mechanism to acting locally, when possible. Interactions with the snapshot manager by the processes are not considered to be events of the computation.

#### 3.1 Comprehensive Monitoring

Let us assume that we desire to continuously
monitor all processes involved in some distributed computation. We are interested only in the global state between transactions. Thus, it is reasonable for processes to incrementally store information locally that is needed by the snapshot manager, and then to send it once, after a *mark* event, a terminal event in a transaction.

Although a computation may be well formed, the snapshot manager may not see all of the *mark* events associated with the computation in a timely manner due to message delivery delays. However, the FIFO nature of the communication ensures that we can extract a well-formed computation from the *mark* events that the snapshot manager knows of. *Mark* events are maintained in a queue of messages from a process until the event can be used to reconstruct a transaction. Reconstruction of a transaction is possible once the *mark* events for all of the processes that participated in the transaction have arrived. The snapshot manager keeps the unused *marks* in queues to ensure that it reconstructs the transactions in the order which they occurred in the computation. In this way the snapshot manager constructs a well formed computation that is a prefix of the actual computation.

To compute the global snapshots the snapshot manager needs to have the process’s state information after the transaction, and be able to determine transaction ordering and transaction membership. The snapshot manager has the process’s state information available because every process sends its monitored attributes to the snapshot manager when the process completes a transaction. The state information is also tagged with the transaction that generated it. A transaction can be uniquely identified by the process id and the sequence number of the *mark* event that ended the transaction. Given the transaction membership of all transactions, transaction ordering can be inferred. If the transaction memberships are known and communication is FIFO, the order that *mark* events are received from a process represents the order in which their respective transactions occurred. This provides sufficient information to reconstruct the transaction ordering.

To compute transaction membership, we require each *mark* to include in its report to the snapshot manager the identity of all the processes which messages have been sent to or received from since the previous *mark*. These are the process’s neighbors during the transaction. The FIFO communication and the *mark* events’ reporting of the process’s neighbors during the transaction allow the snapshot manager to determine the transaction membership. The snapshot manager does this by looking at the earliest mark event from each process not yet associated with a transaction. We know that this *mark* must be part of that process’s next transaction in the computation because of the FIFO communication between the process and the snapshot manager. The snapshot manager then takes the transitive closure over the neighbors reported by the *mark* event. Thus we find the smallest set of *mark* events that have each other’s processes as their neighbors.

### 3.3 Selective Monitoring

As the number of processes grows, it becomes increasingly difficult to monitor all of them continuously without experiencing severe performance penalties. Moreover, the style of interaction associated with query-based visualization entails gradual refinement in the scope of the visualizations and frequent changes in the focus of attention. These, combined with the need to allow the snapshot manager to keep up with the underlying computation and perform long term monitoring, suggest the need for a lighter weight method of collecting snapshots.

To achieve this the user issues queries. A *query* describes a portion of the state that is of interest. The user can use several queries simultaneously to describe what is to be monitored, referred to as the *query set*. The query set used to construct the current snapshot is called the *active query set*. Ideally only processes that are involved in the evaluation of the active query set should be communicating state information to the
snapshot manager, or even be aware that monitoring is taking place. Unfortunately, it is not possible to simply have the mark events of the monitored processes report to the snapshot manager. Consider, for example, Figure 3 which depicts 6 processes and 3 transactions. The processes A, C, and F are the only ones being monitored. Transaction \textit{t2}, which involves only processes B and D, sends no information to the snapshot manager because neither of its participating processes are being monitored. For this reason, the snapshot manager can no longer reconstruct the ordering relation between \textit{t1} and \textit{t3}. Moreover, in \textit{t3} processes C and F have no direct interactions and, since D and E are not required to report to the snapshot manager, it becomes impossible to solve the membership problem for the transaction \textit{t3}. In order to reconstruct the transactions the reporting mark events must supply additional information.

The task of the snapshot manager now becomes one of computing a viewer-specified projection of the global system state. In essence, the snapshot manager filters out the state of the unmonitored processes from the snapshot. It computes the minimum amount of state information that will satisfy the user’s queries.

A transaction is monitored if and only if it contains an event that is monitored. During selective monitoring, the snapshot manager must be able to reconstruct the membership and the happened-before relation for monitored transactions. Since some processes may not report, we cannot use the same solution that we did for the comprehensive algorithm. To address the membership question, we require mark events that are part of the same transaction to perform the logical equivalent of a barrier synchronization during which membership information is collected. To find the membership of a transaction each process performs two actions:

1. Assume the existence of a total ordering over processes. Wait to receive information from all higher-id neighbors, either directly or from another process. Recall that the neighbors of a process are those processes that it communicated with during the transaction.
2. Send all of the known membership information received about other processes who participated in this transaction, including the information received from other processes, to the process with the highest id lower than its own. If there is no process with a lower id, then this process is the transaction leader.

When the transaction leader has completed action (1) it will know the membership of the transaction and can transmit it to the snapshot manager. A more thorough discussion of this can be found in [3].

4. Visualization

The previous sections have described how logically consistent snapshots can be collected from a distributed computation. Equally important is the presentation of the data. The visualization component of the query-based visualization system provides a visual representation of the program’s state in the form of consistent snapshots. Interaction with these visual representations provides an accessible and intuitive mechanism for exploring the state space of the computation.

We use Pavane [8, 9], a declarative visualization system, to present the data. In Pavane, visualizations are implemented as mappings from the state of the program to a collection of graphical objects in a four-dimensional space—the three spatial dimensions plus time. Both the state of the underlying computation and the final set of four-dimensional graphical objects are represented as collections of tuples. They are called, respectively, the state space and the animation space of the visualization. The overall mapping from state space to animation space can be decomposed into a pipeline of any number of sub-mappings, with each intermediate mapping transforming one space into the next in the pipeline; each of these intermediate spaces is also a collection of tuples. Pavane assumes that the underlying computation progresses by means of a series of atomic transitions that modify the state. After each transition, the visualization rules are re-applied to the new state and the resulting animation space is rendered. The snapshot manager acts as the underlying computation that is visualized by the Pavane mapping, providing a centralized view of a distributed computation. The snapshot manager exports to the visualization system the snapshots that it collects. These snapshots are sets of tuples, the process attributes from the computation. Pavane's notion of transaction is consistent with how we view transactions in query-based visualization.

4.1 Animation

Pavane's graphical model provides one or more three-dimensional "worlds", each containing a collection of graphical objects. The animator defines one "window" for each world; the window definition includes the world's properties (center, scaling, background color, etc.), the properties of the display window (dimension, position, etc.), and the types of transformations that the viewer is permitted to make. Subject to any such limitations that the animator requires, the viewer can examine each world "through" its window from any point in the world's coordinate system. The ability to use multiple windows is especially convenient when the effectiveness of two or more visualizations is being compared or when
multiple properties of the same program are to be examined together. Multiple windows on the same state are synchronized to provide a simultaneous view of the state from multiple perspectives.

The use of declarative visualization integrates well with the consistent snapshots collected. We expect the rapid-prototyping success that Pavane has had to carry over into the writing of custom visualization mapping rules as the part of the exploration process.

4.2 Exploration

The notion of navigating through the state space of a computation implies that there is a location associated with the user. There are several ways that the user’s position can change. By default, time advances to reflect the completion of a transaction, with the resulting changes to the monitored attributes. The user can also change his position by issuing a new query in addition to the current specified query set or to replace queries present in the specified query set.

The view that the user has of the computation is also malleable. The user can navigate through the graphical representation of the global state by changing the current point of view or the magnification, or by opening additional windows on the graphical scene. The user can add, delete, and modify the visualization rules to change how the computation is represented.

These changes to the composition of the subset of the computation that is monitored and to the representation of the state of that monitored subset can be achieved in several ways. One possibility is through direct manipulation of the graphical scene. A window containing links to standard and custom visualizations that are relevant to the current location can be used to modify the current set of visualization rules.

In navigating the state space of the computation it is helpful to use both standard and custom visualizations. Standard visualizations provide a familiar framework through which the user may approach the computation. Custom visualizations provide the opportunity to then obtain more detailed information by submitting application-specific queries and rules.

5. Related Work

Many of the ideas embodied in the query-based visualization system we are developing have their roots in our prior experience with the Pavane [8, 9] system and the PARADE [11] and Falcon [2] projects. The Pavane project entailed work on the visualization of concurrent algorithms carried out in conjunction with research on formal design methods. PARADE [10, 11] is a comprehensive environment for developing visualizations and animations of parallel and distributed programs. Our work on this project focused on the development of the Animation Choreographer [5] component, a graphical, interactive tool that supports the calculation of alternate feasible event orderings of the program execution under study, and the visualization of these parallel or distributed programs under a variety of temporal perspectives. Users may select from among default orderings, explore the set of alternate feasible orderings, and display default or custom visualizations of the program’s execution under the selected ordering. These alternate orderings, or temporal perspectives, can provide the user with additional insight into the execution of the program under study. Other components of the environment include POLKA, an animation toolkit, and libraries of views for various classes of programs including PVM and Conch distributed application programs, parallel C threads programs, and HPF high performance FORTRAN programs.

Falcon[2] is a set of tools that collectively support on-line program monitoring and steering of parallel and distributed applications. Components of Falcon include a monitoring specification mechanism, which consists of a low-level sensor specification language and a high
level view specification language, mechanisms for on-line information capture and analysis, mechanisms for program steering, and an associated system for the construction and use of graphical displays of program information. Falcon uses a filtering mechanism to provide on-line event ordering for consistency, supporting interactive steering at appropriate points in the program's execution.

Other related systems include XPVM [4], PGPVM [13], and Ariadne [7]. XPVM provides a graphical interface to the PVM console commands and information, and several animated views for monitoring the execution of PVM programs. The instrumentation to collect the information necessary to produce these standard views is a component of PVM's built-in tracing facility. PGPVM focuses on integrating visualization support directly into a distributed computing system. However, PGPVM does not support interactive exploration or steering. The Ariadne debugger for pC++ programs supports an exploratory approach to debugging a parallel program and emphasizes the integration of event- and state-based strategies. Ariadne provides static displays rather than animated displays.

Our strategies for efficient global state collection rely on related work with snapshot algorithms. An introduction to the global snapshot problem can be found in [1] and an overview of algorithms for dealing with the snapshot problem under different levels of delivery guarantees is found in [6].

6. Conclusions

This paper presents a new program visualization paradigm called query-based visualization. It is envisioned as the basis for a powerful exploratory tools in the hands of programmers and designers working with distributed systems. As the name suggests, the approach combines two key ideas: (1) the notion of treating a distributed computation as an evolving distributed database of objects where state can be examined via queries, which are continually evaluated and (2) the use of three dimensional fully animated images that convey the query results and allow viewers to formulate queries without resorting to textual specifications. The two ideas could be investigated separately but, it is our contention that only by combining them we will be able to have a significant impact on software development practices. An initial prototype employing query-based visualization demonstrates the technical feasibility of the approach.

Acknowledgment: This paper is based upon work supported in part by the National Science Foundation under Grant No. CCR-9217751. Any opinions, finding, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. Bibliography