Interactive Exploration in Virtual Environments

Belleman, R.G.

Publication date
2003

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 5

Interactive dynamic exploration environments*

"Computing machines can do readily, well, and rapidly many things that are difficult or impossible for man, and men can do readily and well, though not rapidly, many things that are difficult or impossible for computers. That suggests that a symbiotic cooperation, if successful in integrating the positive characteristics of men and computers, would be of great value. The differences in speed and in language, of course, pose difficulties that must be overcome."


5.1 Introduction

The interactive dynamic exploration environment (IDEE) model illustrated in Figure 1.2 (page 5) is equivalent to a common design pattern known as the Model-View-Controller (MVC) architecture (see Figure 5.1) [85]. The MVC architecture has its roots in Smalltalk-80 where it was originally applied to map the traditional input, processing, and output tasks to the graphical user interaction model [136]. This three-way abstraction separates (1) the model of the underlying application, (2) the representation of this model to the user and (3) the ways in which the user interacts with the application. Compared to our IDEE model, the "Model" in the MVC architecture corresponds to our Process (or Simulation) component, "View" corresponds to

Presentational and "Controller" to Interaction.

The rationale behind isolating functional components from each other is two-fold. First; this design helps the application engineer to understand and modify each component without having to know everything about the other components. Here, the Model is the central component of the application, the one that does the interesting work. It is kept distinct from the View component which provides methods for the representation of the data structures maintained by the model on a presentation device (e.g. a visual or audio display). The Controller provides an interface from the interaction devices used by the user (e.g. the mouse, keyboard) and sends messages to the model to change its state or to the view to change the representation of the model.

Second, and more important for the environments we are considering here; this design facilitates the distribution of functional components over specialized computing resources. If designed and implemented properly, such a distributed system could provide a more responsive interactive system as compared to the situation where all components execute on the same resource. The separation of an interactive application into components requires some form of communication for the exchange of messages. The implementation of this message exchange mechanism raises additional concerns when the components are distributed over different systems.

5.2 High performance interactive simulation

The responsiveness of an interactive system is directly related to the rate at which updates are generated by each of the components in the system. To increase responsiveness, the delays between the consecutive components in the interactive system should be minimized. In an ideal system, each component produces results the moment input data is available and communication between components is instantaneous. In practice, however, there will always be some delay. The accumulation of all delays is referred to as "update time". In an interactive system there will always
be some delay from the moment interaction takes place until the moment that the environment has reacted to this interaction. This delay is referred to as “response time”.

### 5.2.1 Update and response time

In a non-interactive environment, the update time $T_U$ is the sum over the execution time for the different components ($T_{\text{sim}}$ for simulation, $T_{\text{vis}}$ for visualization and $T_{\text{ren}}$ for rendering) and the communication delay between components ($T_{\text{sim} \rightarrow \text{vis}}$ and $T_{\text{vis} \rightarrow \text{ren}}$):

$$T_U = T_{\text{sim}} + T_{\text{sim} \rightarrow \text{vis}} + T_{\text{vis}} + T_{\text{vis} \rightarrow \text{ren}} + T_{\text{ren}}.$$  

(5.1)

Decreasing update time means that the delays imposed by the different components must be minimized. In the case of executing components this means that the time between the acceptance of input data and the production of results should be minimized. In the case of communication between components, the dominating factor for delay is the time that is required to transfer data from one component to the next.

In an interactive system, the response time $T_R$ depends on which component the interaction is directed to, since only this and subsequent components need to be updated. In case the user interacts with the simulation component, the response time will be $T_R = T_{\text{r(sim)}} + T_U$, where $T_{\text{r(sim)}}$ is the delay between the moment an interaction with the simulation component was initiated and the moment it is received by the simulation.

### 5.2.2 Pipelined execution

A dynamic system differs from a static system in that the simulation component is an iterative process that repeatedly produces (intermediate) results. Basically, the delay at which these intermediate results become available is given by equation 5.1. Figure 5.2 shows a time-frame diagram in a “lock-step” IDEE. In this strategy, the simulation is allowed to advance only if the user explicitly tells the environment it is alright to do so. In this case we say the exploration system is “user driven”. While the user is exploring the results rendered by the graphics system, the simulation and visualization modules sit idle. In situations where a single simulation, a single visualization and a single rendering time-frame takes a negligible amount of time, this strategy may be perfectly adequate since the user will see the result of his interaction in short notice. However, if these time-frames are long, it may take a long time before the result of an interaction is shown. This can lead to an unusable environment and frustration with the user.

If the simulation component is (or can be considered as) a component that is independent of the execution of subsequent components, the update time of the whole environment can benefit from a pipelined execution model. In this execution model, a component resumes execution as soon as its output data has been accepted by the next. The time required before simulation updates are presented to the user (i.e. the length of a time frame on the exploration level) is shortened by allowing the simulation to run in parallel to the rest of the environment, as illustrated by the time-frame
Interactive dynamic exploration environments

Figure 5.2: Time-frames and delays in a lock-step, user driven interactive dynamic exploration environment (IDEE). Time-frames are illustrated by rounded boxes. The gaps between time-frames on a same level represent the idle time of the component on that level. The gaps between time-frames on neighbouring levels represent the delays that occur between the time one component is done with a time-frame and the next component starts working on it. These delays are delineated at the bottom of the figure.

diagram in Figure 5.3. In this case simulation, visualization and rendering execute in parallel. Once all components in the pipeline have executed at least once, and provided sufficient resources are available to execute all components simultaneously, the update time becomes

\[ T_U = \max(T_{\text{sim}} + T_{\text{sim-vis}} + T_{\text{vis}} + T_{\text{vis-ren}} + T_{\text{ren}}). \]  

(5.2)

The frequency of updates perceived by the user at the exploration level depends on the component that requires the longest amount of time to process one time-step. In the case illustrated in Figure 5.3, the simulation is the longest executing component in which case we say that the system as a whole is “simulation driven”. As the components in this execution model are allowed to proceed after having processed one time-step, a situation results where after some time each component is processing a different time-step. Specifically, the user at the exploration level looks at information that was calculated by the simulation component “in the past”. In case the user performs no interaction with the data this is perfectly fine. However, if the user does interact with the data and if this interaction has a direct influence on earlier components (such as the simulation), the component with which the interaction took place may have to rollback to the time step with which the interaction took place in order to obtain consistent results. We will come back to the consequences of this situation when we discuss time management in section 5.3.4.
5.3 Distributed simulation and visualization

The capabilities of modern computer systems may, in some cases, allow both the simulation and visualization components to be performed on the same machine. However, a performance increase may be attained by running these components on dedicated computing platforms. For example, many simulation applications perform better on dedicated hardware such as vector processors, massively parallel platforms or other high performance computing machinery. State-of-the-art graphical systems are now available that are well suited for the rendering tasks. Moreover, a decomposition of an IDEE into separate communicating components facilitates implementation and allows more control over the performance of the system as a whole. However, the decomposition of an IDEE over distributed systems has a number of implications that need to be addressed to obtain a usable exploration system, as described next.

5.3.1 Execution environment

Especially in the case of distributed environments, some means of job control is needed that allocates the resources required for the application prior to execution (when, for example, execution needs to take place on batch queue execution systems). In many organizations such an environment will need to adhere to on-site authentication and authorization regulations as stipulated by the local organization. Furthermore, a mechanism should be provided that allows the user to start/stop the execution of the different components of the environment on the various computing platforms with minimal effort. The application developer cannot expect that an executable and a startup script handed to a user can be successfully used to deploy this program, especially not when distributed execution is involved. Failure is almost certain and it can take days to track down and fix problems. "Globus" is one example of a software infrastructure for distributed computation that integrates geographically distributed computational and information resources [81]. Globus provides a "single sign-on"
service that requires a user to be authenticated only once. After this, the user may execute jobs on any remote system for which they have been given authorization.

5.3.2 Data distribution

In distributed systems, components execute on different, possibly heterogeneous computing platforms. To be able to communicate data with each other, components provide access to their attributes, which can then be made available to other components. In heterogeneous computing environments, the attributes often have to be converted into different representation formats. Furthermore, in many circumstances not all components in an environment will participate in a communication. For these situations a publication and subscription mechanism needs to be provided that limits communication to members of a restricted group. Message passing systems such as PVM and MPI support this data distribution facility for the most part [80, 229]. In general, however, these systems do not support the construction of asynchronous systems as they restrict the application programmer to the Single Program Multiple Data (SPMD) communication paradigm, which in an IDEE would be too restrictive. In most of the interactive simulation systems we are interested in, the data volumes that are generated by the simulation in each time-step are in the order of tens to hundreds of megabytes per time-step. For this reason we are particularly concerned with the communication performance provided by a data distribution facility in the case of large data volumes.

5.3.3 Attribute ownership

The behaviour of individual components in the environment is defined by one or more attributes (or parameters) which together define the state of that component. To avoid race conditions in a distributed system, attribute changes (which can be considered events, for example as a result of user interaction) should only be performed by a component that owns the attribute. In some cases it may be necessary to transfer ownership so that attributes can be changed by other components (for example in a collaborative environment where multiple users manipulate the same components).

5.3.4 Time management

An important issue in an IDEE is time management. In some situations it may be appropriate to constrain the progress of one component based on the progress of another. Time management deals with the exchange of time stamped information between components. For an IDEE, the four most time demanding components are; the simulation system, the visualization modules, the rendering layer and the explorer (i.e. the user). Please recall Figure 5.3 in which a time-frame diagram is shown for a pipelined environment. In a pipelined system, each component is allowed to advance to the next
time-frame when it has finished processing and communicating the results of the current time-frame. Different components may therefore be processing different time-frames at the same wall-clock time. In addition, the rate of advance in time-frames is mostly irregular because of hardware, software and human imposed delays. As a consequence, time delays occur when the output generated by one component cannot be accepted for processing by another component immediately. When components depend on the output of an increasing number of other components, the difference between time-frames that is processed by components further apart in the pipeline increases. This has a causality consequence for the user who explores the final component in the pipeline and therefore interacts with presentations derived from a much “earlier” time-frame than what is being processed by the simulation at the same wall-clock time. Time management is responsible for preventing, or otherwise, detecting and resolving this causality violation. Methods for resolving time causality problems have been investigated within our group [176].

5.4 Communication architectures

Interactive distributed simulation environments consist of interconnected communicating components. The performance of such a system is determined by the execution time of the executing components and the amount of data that is exchanged between components. In the following sections, we will look into a number of different architectures that may be suitable for the implementation of IDEEs. We will describe the capabilities of these systems for the implementation of the model depicted in Figure 1.2 (page 4). As noted earlier; most of the interactive simulation systems that interest us generate data volumes that are in the order of tens to hundreds of megabytes per time-step. For this reason, the communication performance of these systems for large data volumes will be analysed in more detail.

5.4.1 The High Level Architecture

The High Level Architecture (HLA) is a technical framework for modeling and simulation. HLA aims to establish a common architecture for simulation to facilitate interoperability among simulations and promote the reuse of simulations and their components [26, 60–64, 165]. A successor to the DIS (Distributed Interactive Simulation) protocol, HLA provides a robust architecture with which distributed discrete event and other types of simulations can be designed. HLA has replaced DIS as the standard technical architecture for all United States Department of Defense (DoD) simulations since 1996 and has been accepted as an IEEE standard (IEEE1516) in the year 2000.

An HLA simulation consists of a federation that is composed of one or more federates that exchange information in the form of objects and interactions (see Figure 5.4). Interaction between federations is controlled by a “Run-Time Infrastructure” (RTI). HLA systems comprise rules which govern how federates and federations are
constructed (see Figure 5.5); an interface specification which governs how federates interact with the RTI; and an Object Model Template which provides a method for documenting key information about simulations and federations.

![Diagram of HLA components](image)

**Figure 5.4: Components in HLA; federates combine to form a federation which is managed by a Federation Executive (FedExec) process. Inter-process communication is performed through services offered by the Run-Time Infrastructure Executive (RTIExec).**

HLA provides solutions to many of the problems and issues described in section 5.3. Specifically, HLA allows data distribution across heterogeneous computing platforms (including message groups), supports a flexible attribute publish/subscribe and ownership mechanism and offers several methods to do time management. The management services provided by the RTI are separated into six groups of functionality:

1. **Federation management** - this includes services for the creation of federations, joining federates to federations, observing synchronization points, federation-wide save and restores, resigning federates from federations and destroying federations.

2. **Time management** - this includes services that implement the policies and negotiations for advancing logical time. A federate may be “regulating” (meaning it is allowed to send time-stamped events), “constrained” (meaning it is capable of handling time-stamped events), “neither regulating nor constrained”, or “both regulating and constrained”. With each federate adopting one of these four policies, continuous time simulations as well as conservative and optimistic discrete event simulations can be constructed, or combinations of these within the same federation, if so desired.

3. **Declaration management** - this includes functionality for the publication, subscription of object instances or attributes.

4. **Object management** - these include the functions for the registration and instance updates at the object production side and instance discovery and reflection on the consumer side. Object management also includes methods for sending and receiving interactions, controlling instance updates based on consumer demand.
5.4 Communication architectures

**Federation Rules:**

1. Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).

2. In a federation, all representation of objects in the FOM shall be in the federates, not in the run-time infrastructure (RTI).

3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.

4. During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification.

5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

**Federate Rules:**

6. Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).

7. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.

8. Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.

9. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.

10. Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

*Figure 5.5: Governing rules for federations and federates in HLA.*

5. Ownership management - controls which federates are allowed to change (attributes of) object instances.

6. Data distribution management (DDM) - the RTI acts as an “intelligent switch”, matching up producers and consumers of data based on their declared interest. DDM provides a way to further isolate publication and subscription interests by defining a “routing space”, defined by regions.

From the considerations described in the previous sections, the High Level Architecture (HLA) seems to be a suitable architecture for the construction of an IDEE. Figure 5.6 illustrates how an IDEE can be implemented using HLA. This IDEE consists of a simulation federate executing on a High Performance Computing (HPC) system which allows the simulation to run at best performance, and a visualization
and interaction federate that both run on a VR system. All federates are joined into one federation which is governed by a FedExec process. This FedExec process is created by the RTIExec process (which needs to be started manually) the moment the first federate is created and if the federation does not already exist. If the federation already exists, federates simply join the federation.

The federates in the federation communicate using the services provided by the RTI. In this IDEE, the simulation federate publishes simulation results using HLA’s declaration and object management services. The visualization federate subscribes to the simulation results so that these are reflected as soon as results are available from the simulation. Interaction with both the visualization and the simulation federate is performed using HLA’s object management services. Concurrent access to the simulation results is controlled by HLA’s ownership management services while the time management services can be used to control time-frame advances. Both lock-step and pipelined execution models can be achieved with this IDEE through HLA’s time management services; by defining the interaction federate as “regulating” and the simulation as “constrained”, a lock-step execution model is obtained as illustrated in Figure 5.2. By defining the visualization federate as “regulating” and the simulation as “constrained”, a pipelined execution model is obtained as illustrated in Figure 5.3.

![Figure 5.6: Structure of an IDEE using HLA; in this case, a simulation federate is running on a High Performance Computing (HPC) system, a visualization and interaction federate on a VR system. All take part in one single HLA federation, governed by a FedExec process. Communication between federates in the federation is performed through services offered by an RTIExec process.](image)

**Communication performance**

Our primary concerns with HLA are with the performance of communication in HLA and the engineering effort imposed on developers. We have measured the performance of HLA by measuring the transfer time of various sized data blocks between two workstations. Both workstations are based on Pentium III processors, running Linux (RedHat 7.2) and connected via a shared 100 Mbit/s network, using both RTI
1.3NG version 3 and version 5 developed by the Defense Modeling and Simulation Office (DMSO) [60]. Details on the software architecture used for these measurements are described by Zhao et al. [269]. Each measurement in Figure 5.7 is the average over 15 samples. The deviation in the measurements is around one percent.

![Figure 5.7: HLA communication performance between two Pentium III based Linux systems, connected by a 100 Mbit/s network. All measurements are averaged over 15 samples. The deviation in the samples is around one percent.](image)

Internally, RTI uses TCP/IP sockets as its communication mechanism and we can see that the additional services provided by RTI do not put a severe penalty on communication, except for an additional latency for small messages of around 0.02 seconds. For packages sizes less than 512 bytes, we can see that the transfer time is more or less constant. It is very likely that the designers of this RTI implementation decided to optimize the communication of small sized messages, which is probably quite common in the type of simulation systems used by the Department of Defense (i.e. combat simulations). From package sizes of 64 kB and up we can see that the transfer time grows linearly and stabilizes at a throughput of around 1.2 Mbyte/s for RTI version 3 and 5.0 Mbyte/s for version 5. The increase in performance from version 3 to version 5 is significant. Little is known about the communication layer used in the RTI implementation provided by DMSO, but as they were involved with the design and development of CORBA, it could well be that the RTI makes use of “TAO”, a real-time distributed Object Request Broker used in CORBA [270][207,248].

**Engineering an HLA application**

All objects and interactions between objects in a federation must be described according to an “Object Model Template” (OMT). This prerequisite forces the application
engineer to document his code through a formalized framework prior to (and during) implementation. By doing so, applications developed in the HLA framework can be automatically checked for correctness during development and at execution time. The OMT is used to document objects and interactions in a federation in a "Federation Object Model" (FOM), federates are documented in a "Simulation Object Model" (SOM), objects and interactions that are used to manage a federation are documented in a "Management Object Model" (MOM). The development of a SOM (for each federate in a federation) and a FOM (for each federation) is a requirement put forth in the "Federation Development and Execution Process" (FEDEP) model. To help the developer in the formulation of SOMs and FOMs, the "Object Model Development Tool" (OMDT) was developed by DMSO.

The engineering enforced by the FEDEP requires a substantial investment from the application developer. Although this design model forces the engineer to formally describe his code, which is intended to increase the quality of the end product, the initial investment, in terms of time, for the development of new federates is high. However; once defined, SOMs allow simulation components to be reused, which significantly reduces development time.

Conclusions

HLA is a complex but complete framework for the development of interactive distributed simulations. HLA provides most of the services described in section 5.3 that are needed for the construction of interactive distributed exploration systems. DMSO's RTI provides no execution environment that allows HLA simulations to be bootstrapped autonomously; each host participating in a federation requires that an RTI execution environment is started manually. Work is currently underway in our department to see whether HLA can be used in ways that avoid this. Although early version of DMSO's RTI showed communication performance results that were marginal, the performance of recent versions has increased significantly. The latest (and final) version of DMSO's RTI (version 6) was released on September 30, 2002 [67].

5.4.2 SPLICE

SPLICE is a software architecture developed at Hollandse Signaalapparaten B.V. (HSA) for large-scale distributed embedded systems [17, 18, 59]. Included in the design goals of SPLICE was the aim to provide an architecture that reduces the complexity of the development of large, reactive distributed systems, to provide fault-tolerance and to allow incremental deployment and development of systems. SPLICE uses a data-oriented coordination model in which multiple processes exchange messages. Each process has its own communication "agent" that maintains a "local store" (see Figure 5.8). As in HLA, there is no explicit communication between processes. Instead, each process executes autonomously and the communication agents manage communication between distributed processes based on a subscription paradigm (hence the acronym; Subscription Paradigm for the Logical Interconnection...
of Concurrent Engines). Each process publishes and/or subscribes to one or more "data sorts" that identify messages. When a process publishes a datum, its agent will store it in the local data store and forward the datum to all agents that manage a process that has subscribed to this datum. The data space that is thus created resembles the "tuple space" paradigm used in Linda [214]. As all processes work independently from each other, global control is minimized which greatly reduces the complexity of designing a distributed reactive system.

![Figure 5.8: Components in SPLICE: processes, each with an agent and a local store that communicate over a network.](image)

SPLICE makes a distinction between periodic, context and persistent data to handle different communication needs between processes. Periodic data sorts are used in situations where data is generated repetitively and the validity of this data decreases over time. Data in this category is delivered only to processes that are active at the time it is published. Context data sorts are used to represent the current state of the system as a whole. SPLICE ensures that the most up-to-date values are always available and provides all context data to each newly created processes. Persistent data is like context data except that storage is non-volatile. Sorts that are labeled "persistent" are stored in a persistent database even after processes have gone offline. When new processes require this data, the values are retrieved from the database. Processes may identify fields in the data structure of a data sort as "key-fields". New instances of a dataset that are identical in all key-fields replace earlier produced data while all other instances will be stored separately. Even if the data structure in a dataset is the same for both producers and consumers, the key-fields may be different. This mechanism allows subscribers to index data that was produced while it was unable to consume the data.

Figure 5.9 illustrates how an IDEE can be implemented using SPLICE. This IDEE consists of a simulation process executing on a High Performance Computing (HPC) system which allows the simulation to run at best performance, and a visualization and interaction process that both run on a VR system. All processes communicate through local communication agents provided by SPLICE. In this IDEE, the simulation process publishes simulation results as "periodic" data through its local agent.
The visualization process subscribes to the simulation results so that these are reflected by its local agent as soon as results are available from the simulation. By providing different “key-fields”, separate time-frames of simulation results are maintained to avoid that simulation data is missed by the visualization or interaction processes in case they were unable to consume the data. Interaction with both the visualization and the simulation processes is performed using the same publication and subscription methods. Although SPLICE provides no explicit services to support ownership management and time management, the existing services can, in principle, be used to implement these.

**Communication performance**

As SPLICE is primarily designed for the exchange of small messages between many processes executing at the same time, we were concerned with its performance for the communication of large data volumes. To determine this, we designed and implemented a program using SPLICE that exchanges large messages between two distributed systems. Figure 5.10 shows the communication performance of SPLICE (version 3.8) measured between two Sun UltraSparcs, connected by a 100 Mbit/s network connection. Each measurement is averaged over 15 samples.

The measurements show that the increase in transfer time differs below and over a message size of 1 Mbyte. At this message size, the transfer rate is approximately 500 kByte/s. For larger messages, the transfer time increases at a steeper rate and trans-
Figure 5.10: SPLICE communication performance for large messages between two Sun connected by a 100 Mbit/s network connection. Each measurement is averaged over 15 samples.

Transfer rate drops back to approximately 200 kByte/s for a 10 Mbyte message. It turns out that this is related directly to the amount of shared memory available on a system. SPLICE stores data into shared memory to improve communication between processes on the same system. However, it also uses this memory as a buffer for communication to processes on other systems by the communication agents. Unfortunately, shared memory is a limited resource on most operating systems. When the amount of shared memory is depleted, SPLICE is forced to break up messages into smaller blocks. This, as well as the overhead imposed by the exchange of multiple blocks for one message, reduces the communication throughput for large messages.

Conclusions

The paradigm provided by SPLICE allows for the development of large, complex distributed systems with relative ease. It is clear that HLA and SPLICE share many of the same design decisions, however, the construction of distributed systems using SPLICE is much easier. SPLICE provides most of the services described in section 5.3 that are needed for the construction of interactive distributed exploration systems. However, the implementation we used for our performance measurements showed that SPLICE is less suitable for systems that exchange large data volumes.
5.4.3 The Virtual Laboratory and the Data Grid

VLAM-G, the Grid-based Virtual Laboratory AMsterdam (or VL for short), provides a science portal for distributed analysis in applied scientific research. It offers scientists the possibility to carry out their experiments in a familiar environment, where content and data are clearly separated. Emphasis is put on the development and use of open standards and seamless integration of external computational resources across organizational borders [3]. A modular architecture is essential to establish such a flexibility. Supporting inter-disciplinary interactions implies the composition of (software) modules, which may have been developed independently. VLAM-G takes this modularity into account by providing a modular data flow-based system. In order to ensure that only proper connections can be established, the constituting modules communicate using a strongly typed communication mechanism.

A common Run-Time System (RTS) has been designed which can properly deal with a set of generic modules for various scientific areas. A preliminary requirement analysis performed together with various scientists from the three application domains has led to a classification of three types of modules: control modules that control external devices, processing modules that perform data filtering and visualization, and database and mass storage modules that allow the experimenter to store and retrieve local and remote data, either in raw data files or in databases. A scientist assembles these modules to compose his experiments. VL middleware assists the scientist by providing a Graphical User Interface (GUI) and an assistant. These components ease the handling of remote data access, resource allocation, security issues and access to external devices. The editing phase of an experiment is performed using a portal, implemented as a drag-and-drop GUI.

Communication in VL

Modules in VL start execution when consumable data is available and produce results for other processes to consume, much in the same way as in visualization pipelines described earlier in section 1.3.2 (page 7) [262]. The main difference with SPLICE and HLA is that the communication path from one process to another is explicit in VL. Processes in VL are logically connected in Process Flow Templates (PFT) that identify (1) the modules that are part of an application (a so-called “experiment” in VL terminology) and (2) the logical connections between the modules (see also Figure 5.11). So-called “portals” provide access to the different aspects of VL for the definition of experiments, launching experiments, data storage of intermediate results and visualization front ends for (intermediate) representation of experimental results. Communication between modules is performed by the Virtual Laboratory Abstract Machine (VLAM) which uses the Globus communication primitives to transfer data across organizational borders [81,88].
5.4 Communication architectures

Figure 5.11: An example of an experiment definition in the Virtual Laboratory. This example shows an IDEE where resources from different institutes are combined into an experiment for interactive flow simulations through medical scans (see also Chapter 6).

Concluding remarks

The design of VL addresses some of the issues described in section 5.3 for the construction of an interactive dynamic exploration environment (IDEE). In particular, VL provides an execution environment that hides many of the details of the execution of distributed processes on different systems across organizational domains. The communication and execution mechanisms provided by VL are sufficient for the implementation of a pipelined distributed IDEE as described at the beginning of this chapter, although they do not provide the richness of capabilities offered by both HLA and SPLICE. Unfortunately, the performance of the communication mechanisms in current implementations of VL is inadequate for the exchange of large data volumes (data not shown).

5.4.4 Concluding remarks on communication architectures

The communication architectures just described address many, though not all, of the design criteria for the construction of an IDEE that were set forth in section 5.3. HLA (the High Level Architecture) provides solutions to many of the problems and issues described in the previous sections and is, therefore, a suitable architecture for constructing an IDEE. Specifically, HLA allows data distribution across heterogeneous computing platforms (including message groups), supports a flexible attribute
publish/subscribe and ownership mechanism and offers several methods to do time management. The use of HLA in interactive simulation systems is described in more detail by Zhao in [268, 269]; this work describes an Agent-based Intelligent Virtual environment (AG-IVE) where agents are used to coordinate communication between distributed systems on behalf of processes that take part in an interactive simulation system.

One concern that remains is that of the interchange of large data volumes in an IDEE. Again; as noted earlier, most of the interactive simulation systems that interest us generate data volumes that are in the order of tens to hundreds of megabytes per time-step. In the following section we describe a number of techniques that can be used to increase the throughput of a communication architecture. These mechanisms are used in a test case of an IDEE which will be described in chapter 6.

5.5 High throughput communication using CAVERN

Distributing components over different systems means that some form of communication must be established to allow the output of one component to be transferred to the next. It could be that the overhead generated by this communication mechanism annuls the benefits obtained by the distribution (i.e. although \( T_{c,1} \) decreases through the use of optimized resources, \( T_{c,1\rightarrow c,2} \) increases because of the extra communication overhead between two components \( c,1 \) and \( c,2 \)). To reduce the delays caused by communication it may be beneficial to reduce the amount of transferred data as much as possible. This reduction itself, however, also takes time so careful consideration is often necessary.

We have implemented three mechanisms to increase the throughput of data transfers over a network. These mechanisms are cascaded into a pipeline to decrease the amount of transferred data and spread the remaining data over multiple network connections, in an effort to maximize throughput (see Figure 5.12).

![Diagram](send+receiver.png)

**Figure 5.12:** The stages in the communication pipeline: the data volume produced by a sender is reduced in volume by an application specific encoder and a compression stage. The remaining data is sent through multiple network connection to the receiver where the pipeline reconstructs the data for the receiver.
5.5 High throughput communication using CAVERN

5.5.1 Hiding latency by using multiple connections.

The first method increases communication throughput by using multiple network connections between peers at the same time. There are two reasons why this increases throughput (see also Figure 5.13). First; in the case of reliable network connections (such as connections using the TCP protocol), delays caused by waiting for acknowledgment packets can be “hidden” by serving other connections that are ready. Second; the technique can exploit “intelligent” network devices that spread communication over different routes. This technique therefore performs best when there are many such devices between peers (which is often the case when peers are geographically dispersed). In principle, data can travel along different routes from peer-to-peer, thereby circumventing congestion caused by other traffic on the network.

![Diagram of network connections](image)

*Figure 5.13: Increasing communication throughput using multiple network connections.*

5.5.2 Data volume reduction

Note that the total volume of data that is transferred between peers is not decreased by the method described in the previous section. Instead, it increases throughput by exploiting as much available bandwidth as possible. The techniques described next aim to increase throughput by decreasing the volume of communicated data.

Data encoding

Data encoding is a semantic volume reduction technique that aims to reduce the volume that is needed to represent the data to a minimum. This technique relies on the fact that the receiving side may not always be interested in the most accurate representation of the data that was calculated by the sender. Because this type of data reduction throws away unnecessary information, this method is frequently referred to as *lossy compression*.

Although a significant reduction in volume can be achieved using this technique, the receiver should be conscious of the fact that the data it has received is not of the same accuracy as was originally produced. Although this reduction may be acceptable
under some circumstances, unexpected side effects may occur when the errors that are present in the data are accumulated due to an integrating method of analysis on the data. For example, an often used technique in vector field visualization is to represent the path of the flow using streamlines. These streamlines are created by integrating over individual vectors. Due to the accumulation of (small) errors in each individual vector, a streamline may follow a radically different path.

**Data compression**

Freely available compression libraries (such as zlib [152]) provide means of reducing data volumes very effectively. This data reduction does not make any assumption about the data contents and is therefore without any loss of information. Most compression libraries can be parameterized to indicate the level of compression that should be achieved at the expense of higher execution time. This type of data reduction is commonly referred to as *lossless compression* as the data is unchanged after decompression. The amount of compression depends on (1) the type of data and (2) the amount of data. In general, the compression ratio decreases when the amount of data decreases. Note that this is important in parallelized applications in which the original data volume is often decomposed into smaller subvolumes.

### 5.5.3 Performance of the network communication pipeline

We have implemented the mechanisms described above using CAVERN [145,169]. In this section we show some results obtained from measurements on their performance. Figure 5.14 shows the mean throughput over 200 measurements of the multiple connection stage in the communication pipeline. These measurements were taken between a system in Amsterdam and St. Petersburg, Russia. Over a single connection, the throughput between these systems was approximately 100 Kbyte/s. As can be seen from this figure, the average throughput increases as more connections are used, but up to a maximum. Using more connections congests the network and no further increase in throughput can be obtained. As more connections are used, the throughput becomes more and more unpredictable and decreases when too many simultaneous connections are used.

To illustrate the influence of encoding on throughput, Figure 5.15 shows the mean performance over 5 measurements of the complete network communication pipeline on the transfer of the results from a parallel lattice Boltzmann flow simulation kernel to a visualization frontend; using compression, multiple network connections, both with and without encoding. This figure illustrates that, on average, encoding doubles throughput. Although this figure shows the typical throughput that can be achieved, in some situations the total performance of the network communication pipeline resulted in a throughput of 62 Mbyte/s, which is over 5 times the bandwidth of the slowest network link (100 Mbit/s).
5.5 High throughput communication using CAVERN

![Average network throughput](image)

**Figure 5.14:** Average network throughput when using multiple network connections between a system in Amsterdam and a system in St. Petersburg, Russia.

![Mean throughput](image)

**Figure 5.15:** Mean throughput of the network communication pipeline when used to transfer the results from a parallel lattice Boltzmann flow simulation kernel to a visualization frontend; shown with and without encoding. Multiple network connections and data compression were used in both measurements.

### 5.5.4 Conclusions

Our measurements show that a great performance increase can be obtained with the mechanisms described in this section. The results presented here were obtained
using unoptimized algorithms; we expect that performance could be increased even more. The different stages in the communication pipeline require a certain amount of time to execute which adds delay to communication time. By tuning the parameters that influence this execution time, throughput can (in principle) be automatically optimized. For example; networks are generally shared by many institutes so that available bandwidth changes over time. The multiple connection technique can sense this change by measuring the effect of employing more or fewer connections during communication. An optimal number of connections can thus be determined dynamically (and transparently) while peers communicate. However, a different parameterization at one stage influences the execution time of subsequent stages which implies that parameter optimization is not a trivial task.

5.6 Summary and conclusions

This chapter addressed the issues involved in the design and implementation of high performance interactive dynamic exploration environments (IDEE). We argued that a design where processing, presentation and interaction are isolated into separate components can facilitate the engineering of an IDEE and allows the components to be distributed over specialized computing resources to increase performance. We described two execution models for IDEEs; a user driven and a simulation driven execution model, and discussed the performance behaviour and functional implications in the use and implementation of these models. The decomposition of an IDEE over distributed computing resources requires a supporting communication architecture. Several communication architectures were described and evaluated; the High Level Architecture (HLA), SPLICE, and the Grid-based Virtual Laboratory AMsterdam (VLAM-G). We analysed the communication performance for the exchange of large data volumes in more detail.

The overall performance of a distributed IDEE benefits from high throughput communication. We designed three independent, cascadable mechanisms to increase communication throughput and measured their performance. These measurements show that a great increase in communication throughput can be obtained with these mechanisms. However, the measurements also suggest that throughput can be improved further by dynamically tuning the behaviour of the different stages at run-time.