The Distribution Subsystem

Erik Klintskog
Anna Neiderud

Version 1.3.1
June 9, 2004

Abstract

Mozart provides support for distributed programming. This is realized by a Distribution Subsystem that offers communication with an arbitrary number of sites, a modular design that allows customization, communication over multiple platforms, and reliable message delivery and fault tolerance; all this with high throughput through an efficient implementation.

To allow application programmers to more efficiently utilize distributed programming an overview of the Distribution Subsystem with references to other documentation is presented by this document.

Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

1 Introduction **1**

1.1 Layered Design . . . 2

1.2 Outline . . . 2

2 Connection Establishment **5**

2.1 Bootstrapping a Connection . . . 5

2.2 Dynamic Connection Establishment through Connect-Accept-Pairs 6

2.3 Automatic Connection Opening and Closing . . . 7

3 Entities and Protocols - a Background **9**

3.1 Distribution of References . . . 9

3.2 Consistency Protocols . . . 10

3.3 Distribution of Mozart Entities . . . 11

3.4 Distributed Memory Management . . . 11

4 Efficient Implementation **13**

4.1 Throughput . . . 13

4.2 Resource Usage . . . 13

4.3 Zero Impact Implementation of the Distribution Subsystem . . . 14

5 Failure **15**

5.1 Handling Faults from Mozart . . . 15

5.2 Fault Detection . . . 16

5.2.1 Permanent Fault . . . 16

5.2.2 Temporary Faults . . . 16

5.3 Fault Tolerance . . . 17

5.4 Perm and Temp on the Internet . . . 17

6 Monitoring Tools **19**

6.1 The Distribution Panel 19

6.2 The Oz Message Tracer 19

6.3 The Mozart Distribution-Subsystem Simulator 20

7 Interfaces **23**

1 Introduction

Mozart provides support for distributed programming. This document describes how distribution is realized and what features Mozart offers to deal with distribution. Guidance on how to practically do distributed programming can be found in the *"Distributed Programming in Mozart - A Tutorial Introduction"*.

Distribution is realized by a *Distribution Subsystem* (DS) used by the Mozart process. The DS implements a state-of-the-art middleware abstracting away the network from higher levels. This is achieved by defining entity consistency protocols that maintain the semantics of local Oz-entities also when distributed. Messages in these protocols are reliably delivered disregarding, transient network congestion and partitions.

Communication with an arbitrary number of sites[1]

Through a 128 bit unique-id schema, billions of remote machines can be addressed. By letting connections time-share the usage of physical channels when there is a lack of resources, the DS is able to handle thousands of simultaneous connections even on off-the-shelf hardware.

Modular design allows customization The DS has a layered and modular design enabling additions of new transport protocols. The DS currently uses TCP, but work is being done to add a lean reliable UDP implementation and a shared memory communication schema.

Platform Independent The implementation of the DS is mature, stable, and available on multiple platforms. Currently, versions for Linux, Windows and many flavours of Unix exist. The protocols used for transferring data are platform independent, enabling inter-platform communication.

Serialization of the language graph An advanced marshaler is available to serialize language graphs to be fit in messages. During marshaling the different types of entities are treated separately to allow replication of stateless entities, and distribution of references to stateful ones. An unmarshaler is able to build a corresponding language graph at the receiver side, replacing remote entity references with proxy structures.

[1] Every Mozart process is referred to as a site.

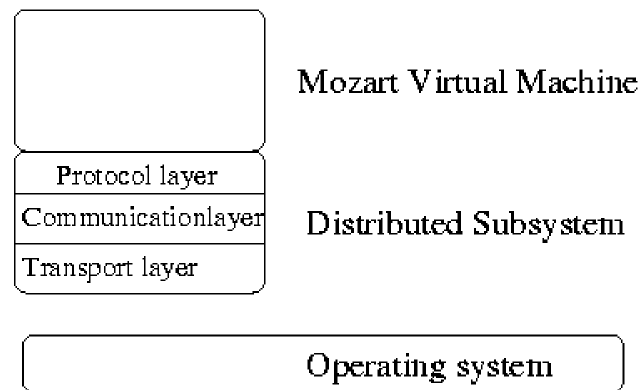
Reliable message delivery and fault tolerant The DS ensures reliable delivery of messages even in the case of transient faults. By use of an acknowledge schema, transient faults can be recovered.

High throughput with an efficient implementation By clever usage of pipelining in bounded memory, together with priority based message sending, the overhead imposed by the complexity of the DS makes the communication only slightly slower than over a raw socket implemented in C.

1.1 Layered Design

The DS is responsible for opening and closing connections, transferring messages, reporting network problems and executing the consistency protocols attached to all distributed entities. This is implemented by a three-layered structure (see Figure 1.1) with the three layers *Protocol Layer*, *Communication Layer* and *Transport Layer*. The topmost layer, the Protocol Layer, is responsible for running the protocols that the DS implements. The middle layer, the Communication Layer, handles virtual channels that are created between different processes. The lowest layer, the Transport Layer, implements an interface to any transport protocol used to transfer messages between machines. The Communication Layer and the Transport Layer together are also referred to as the *Message Passing Layer*. A more detailed description of this design can be found in Anna Neiderud's master's thesis report[2].

Figure 1.1 The Mozart Virtual machine and the Distribution Subsystem



1.2 Outline

The remainder of this document describes some major parts of the DS and their implementation. Chapter 2 explains the bootstrapping sequence for connections and how connection establishment can be customized. Chapter 3 then discusses how entities of

[2] `http://www.sics.se/~annan/ThesisReport.doc`

Mozart are distributed. After that Chapter 4 shows how the implementation has been made efficient. A challenging question in distributed programming is failure, and how to deal with it is discussed in Chapter 5. Chapter 6 and Chapter 7 give useful pointers to understanding and using all the features described.

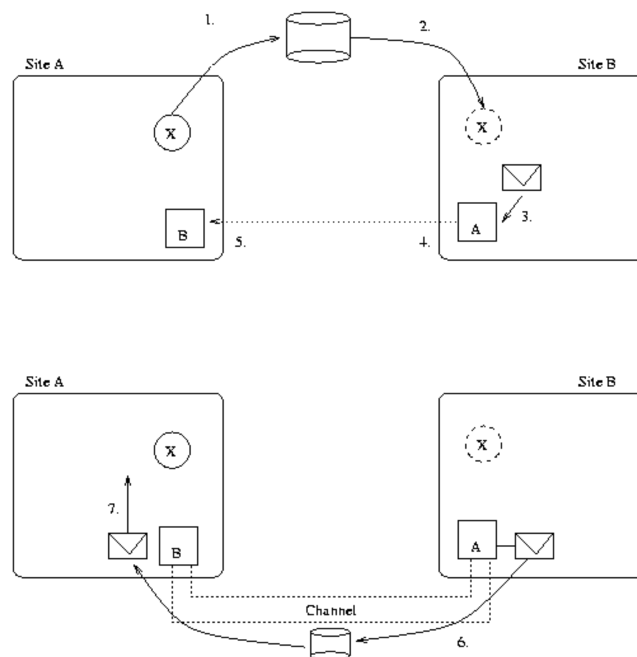
Connection Establishment

When a Mozart site is a participant in a distributed computation, entities are automatically distributed via lexical scoping and connections are opened when needed. Before this can take place an initial entry point in the distributed application must exist. This is what we refer to as bootstrapping a connection.

2.1 Bootstrapping a Connection

The Mozart programmer establishes connections to the outside world by offering a ticket to an entity. Such a ticket is an character string containing enough information for other Mozart sites to connect to the offering site and access the offered entity. This connection establishment procedure is illustrated in Figure 2.1.

Figure 2.1 Bootstrapping a Connection



1. Site `A` offers a ticket to entity X and saves the string to persistent storage available to B.
2. Site `B` loads the string and *takes* this ticket which will cause a representation of site `A` and a representation of X to be created at B.
3. Site `B` creates a message requesting access to X and passes the message down to the representation of `A`.
4. A connection to `A` is requested from the connect-accept-module.
5. At site `A` an incoming request for a connection is accepted. During a hand-shake phase the representation of `B` is passed to `A`.
6. Site `B` marshals the message and sends it to `A`.
7. Site `A` unmarshals the message and passes it to the appropriate protocol message handler.

2.2 Dynamic Connection Establishment through Connect-Accept-Pairs

How a message is transferred between two sites, and how a physical connection is initialized is of no importance to the semantics of entities. However, this is of great importance when it comes to communicating over different types of networks with different restrictions such as security requirements or firewalls. Mozart offers a default connect-accept-module with non-secure initialization of a connection over TCP, but the interfaces are open for the programmer to customize parts or all of the procedure. This is possible through the notion of Connect-Accept-Pairs and the three layer design, allowing for replacement of the transport layer.

Connect-Accept-Pairs To define how a physical connection is initialized, a pair consisting of a connection and an accept procedure can be customly defined. The accept procedure defines how a site accepts incoming requests for connections and runs locally at that specific site. The connection and accept procedure must agree on a scheme to establish a connection, and the connection procedure should in the future be possible to pass around with any reference to the site. Currently an application programmer can create a custom pair and manually install it in all processes involved in a distributed application. This will create a subdomain only accessible to those processes.

Transport Modules A transport module is responsible of delivering messages from one site to another. It may be a very lean layer interfacing TCP or contain a complete implementation of a new transport protocol.

2.3 Automatic Connection Opening and Closing

For efficiency reasons, physical connections should be open while there is a need for them from at least one end, and there are enough resources to maintain them. To achieve this, connections are automatically established when the need arises and closed when the need no longer remains. The latter part is handled by the garbage collector; once the last reference is garbage collected locally, the connection will be closed. When resources are low, connections will be taken down temporarily.

3

Entities and Protocols - a Background

Distribution in Mozart is realized by sharing language entities. How this is done depends on if they are stateless or stateful. Stateless entities are replicated between sites whereas stateful entities create access structures. The local semantics of the stateful entities are maintained in a distributed setting by a number of consistency protocols.

Mozart entities may be in one of two states: plain or globalized. A plain entity is only referenced within its virtual machine. This virtual machine is called the entity's home site. A plain entity will become globalized at the instance when a reference to it is shared. A globalized entity is returned to being plain when no more remote references exist; this is called localization.

The Distribution Subsystem offers four kinds of distribution behavior and four kinds of replication patterns. Distribution behavior and replication patterns are paired together to create different distribution semantics. Each entity type in Mozart has been assigned a pair to define its distributed semantics and behavior.

3.1 Distribution of References

At globalization an entity is given a globally unique identity (GUId). This name is used to identify imported references, to see if a copy of the entity already exists at the importing site. The GUId consists of a reference to the home site and an identifier that is unique at the home site.

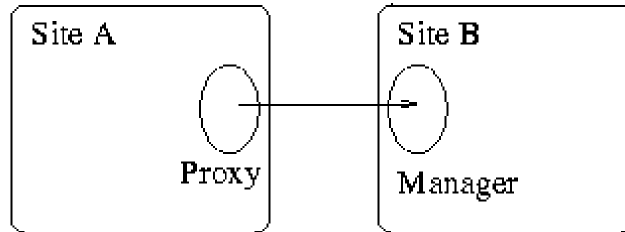
When entities are transferred from one site to another five kinds of replication patterns exist. The importing site is responsible for building a structure according to the replication pattern. The five patterns are described here:

Replicated Replicated entities do not have a GUId. Instead enough information is transferred at distribution time to enable the importer to build a complete copy of the data structure. Without a GUId it is not possible to determine whether the entity already exists at the importing site or not.

Replicated Uniquely An entity that is Replicated Uniquely transfers enough information to build a complete replica of it at the importing site. In contrast to Replicated entities, Replicated Uniquely entities have a GUId that enables the importer to make sure that there may exist one and only one instance of the entity at the importing site.

Access Structure When a stateful entity is globalized, a Manager is constructed at its home site. References to the the entity imported at other sites result in the construction of proxies pointing to the manager (see Figure 3.1). This structure is called an access structure. The GUId of the entity is used to ensure that one and only one proxy is built at site that imports references to the entity.

Figure 3.1 A Proxy at site A refers to its Manager at site B



Access structures are used to maintain reference consistency and as a base for the consistency protocols described below.

Lazy Replication Lazy Replication is a special case of Replicated Uniquely. Instead of sending the whole value of the entity the possibility to build an access structure is transferred. If no instance of the entity exists at the importer, the access structure is built. When a Lazy Replicated entity is accessed for the first time the value of the entity is requested and the access structure is removed when the value arrives.

Resource Placeholder Some data structures should not be available remotely. This is handled by replacing those entities with a placeholder on which no operations can be performed except equality. This placeholder is reflected to language level as a Resource.

3.2 Consistency Protocols

As stated earlier, the DS uses consistency protocols to maintain the semantics of a plain entity also when it has been globalized. Note that this is of course not necessary for replicated stateless entities.

Stateful entities can have a real state or be single assignment, where single assignment means transforming into another entity once. This is used to implement distributed logic variables. Single assignment entities can be dealt with efficiently by a proxy-manager structure where the manager knows and notifies its proxies. Other stateful entities may implement distribution by letting their state move around to active proxies or by letting a manager act as a server for read and write requests. This gives us with three different protocols for maintaining entity semantics:

Stationary State The manager maintains the state of a stateful entity locally, and proxies send read and write requests asynchronously to access the state.

Mobile State Any proxy can attract the state of a stateful entity and operate on it locally as on a plain entity while the state is present.

Single Assignment The manager knows all of its proxies and can administer any request to transform the entity to a reference to another entity, and forward this to all proxies.

3.3 Distribution of Mozart Entities

Figure 3.2 shows the class of reference distribution and the consistency protocol used for entities in Mozart.

Figure 3.2

Entity Type	Reference Distribution	Consistency Protocol
Port	Access Structure	Stationary
Variable	Access Structure	Single assignment
Cell	Access Structure	Mobile state
Lock	Access Structure	Mobile state
Object	Lazy Replication	Mobile state
Record	Replicated	none needed
Atoms	Replicated	none needed
List	Replicated	none needed
Chunk	Replicated Unique	none needed
Name	Replicated Unique	none needed
Class	Replicated Unique	none needed
Functor	Replicated Unique	none needed
Procedure	Replicated Unique	none needed
Code	Replicated Unique	none needed
Dictionary	Resource Placeholder	none needed
Array	Resource Placeholder	none needed
Sited Entities	Resource Placeholder	none needed
Builtin	Resource Placeholder	none needed
Constraint Variable	Resource Placeholder	none needed

3.4 Distributed Memory Management

Every Mozart site performs garbage collection locally. During this process all replicated entities and proxies are treated as plain Mozart data structures. Managers on the other hand act as roots for the local garbage collector.

To ensure that globalized entities are localized if and only if no more remote references exist, the DS has a distributed reference consistency algorithm. This is currently im-

plemented by an extended version of Weighted Reference Counting[1] called Secondary Weight. Weighted Reference Counting is an algorithm that assigns a total weight to an entity. When references are shared, a part of this weight is shared too. When all weight is present at the manager, the entity is local. The original algorithm has a problem in that weight is limited. Secondary Weight overcomes that problem, by allowing proxies to create a new range of weight that they manage.

[1]Presented independently by D I Bevan in *Distributed Garbage Collection Using Reference Counting*,1987, and Watson and Watson in *An Efficient Garbage Collection Scheme for Parallel Computer Architecture*, 1987.

4

Efficient Implementation

The Distribution Subsystem of Mozart boosts throughput and can communicate with an unbounded number of sites in spite of bounded resources such as memory or a limited number of connections. The ability to distribute any entity does not impose any noticeable performance loss on local entities. These properties are achieved by an efficient implementation which will be discussed in this section.

4.1 Throughput

Over most transport media throughput can be boosted by sending few fairly large packets rather than many small ones. The DS utilizes this by pipelining, that is sending several messages to the same destination in one packet. This is possible since all messages to one site are sent via the same (virtual) connection, and since all sends are asynchronous. Asynchronous sends allow a short delay between the time a message is constructed and actually sent. During this period of time, more messages from the same or a different computation may be constructed and can be sent together. [1]

The perceived throughput consists not only of the amount of bytes transmitted, but also of the importance of the transmitted data. Important data is therefore sent with a higher priority. Large low priority messages may also be interleaved by high priority messages by sending large messages in pieces. Note that no messages are lost.

4.2 Resource Usage

The resources that are limited in this context are mainly memory and number of connections. The number of connections available depends on what transfer medium and what operating system is used. For a TCP-connection the limit is imposed by a limited number of file descriptors.

Memory The largest amount of memory necessary for communication is that of the buffers necessary for marshaling. To avoid a memory blowup when sending large messages, the DS uses a suspendable marshaler. This marshaler marshals data to buffers limited in size and suspends in case they are filled. The partial message can then be sent and marshaling can be continued later.

[1] This is a more fine-grained version of the Nagle algorithm.

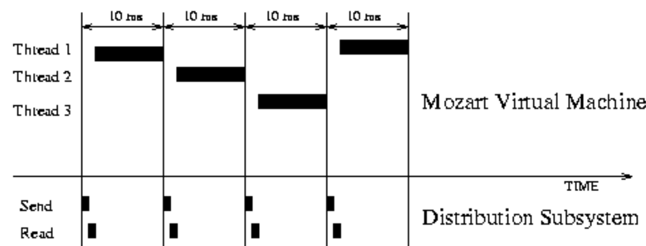
Connections The fact that only one virtual connection per pair of sites is used, limits the number of physical connections. Unfortunately, when resources are scarce, this is not always enough. Some applications may acquire references to a large number of sites without ever communicating with all of these. Other applications may need to do concurrent communication with more sites than can be simultaneously physically connected. The former case is handled by the automatic connection establishment property, the latter by a round robin scheduling mechanism that lets all virtual connections take turns on having one of the possible physical connections on a need to use basis.

4.3 Zero Impact Implementation of the Distribution Subsystem

The Distribution Subsystem does not interfere with the performance of the local Mozart engine. This is achieved in several ways:

- The DS is a dynamically loadable library that will only be loaded into the system when communication is needed.
- All sockets used for communication are non-blocking, enabling the local computation to advance while the network is transmitting data.
- The implementation of entities is done in such a way that the distributed version will only be invoked when necessary.
- The execution time of the operating system process is shared between the Mozart Virtual Machine and the DS. This is achieved by passing messages asynchronously via queues between the two systems, and processing them in each systems timeslice (See Figure 4.1).

Figure 4.1 Time sharing between the Mozart Virtual Machine and the Distribution Subsystem



5

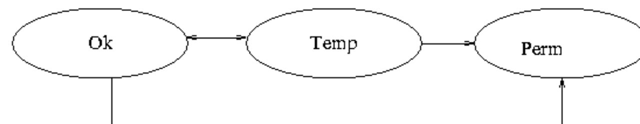
Failure

The Mozart system uses a three-state-model for remote sites. The three states are: OK, Temporarily Lost (Temp) , and Permanently Lost (Perm). At first introduction a remote site is always in the state OK, further use might change the state of the site into one of the other two states. The states are described here:

- OK: No problem has yet been detected on the remote site.
- Temporary Lost: Some transient kind of problems has been detected. A network fluctuation or partitioning can be the cause.
- Permanently Lost: The site is gone and will never be contactable again. The process is gone or the machine is taken down.

Figure 5.1 shows the transition-graph for the different states. Note that Perm is a non-transient state.

Figure 5.1 Transition-graph over OK, Temp and Perm



The classification of site failures into the two categories *Temp* and *Perm*, simplifies construction of fault tolerant applications. By reflecting the failure state of an entity's home site to language level, the programmer is presented detailed information on what to expect from an entity in the future. The interpretation of the state then is: An entity in the state Temp might be accessible again while a Perm state clearly defines that the entity will never be accessible again.

5.1 Handling Faults from Mozart

Mozart offers different ways to instrument the behavior of distributed entities in the case of failure (also see Chapter *Detecting and Handling Distribution Problems:* `Fault`, *(System Modules)* and Chapter *Failure Model*, *(Distributed Programming in Mozart*

- *A Tutorial Introduction)*). A fault condition can be paired together with a reaction strategy implemented by a procedure; this pair can be installed on a particular entity. When a fault occurs that matches the fault condition the reaction strategy is executed.

There are three different ways to instrument an entity's behavior when faults interfere with their usage:

- Fault Exceptions: Fault Exceptions are per default thrown when an operation on a failed entity is attempted. The condition on when an entity is failed is set by a global fault condition.
- Watchers: A Watcher is an asynchronous fault handler that monitors a specific entity. When the fault condition for that entity matches the installed fault condition the reaction procedure is started in a new thread.
- Handlers: An Handlers is similar to a Watcher, but is only triggered when an operation is attempted on the particular entity. If the entity has a fault condition that matches the installed fault condition, the reaction procedure is executed instead of the operation, in the same thread.

5.2 Fault Detection

The Message Passing Layer (the Communication Layer together with the Transport Layer) monitors its channels to detect any kinds of problems. To simplify the design Perm is only detected while opening connections, while Temp is detected on an open channel as described below.

5.2.1 Permanent Fault

Operating system error codes are used to deduce that a remote site is permanently down. The main reason for this is that the error codes that guarantee that a process does not exists are only propagated at connection attempts.

5.2.2 Temporary Faults

Temporary Fault detection is done on Round Trip (RT) calculations. Each Mozart site has an Acceptable Round Trip (ART), when a RT to a remote site is higher than the ART the remote site is defined as Temp. When the RT goes down below the ART the remote site is considered OK again. The ART can be instrumented from language level.

The message passing layer constantly measures the RT to all remote sites it is connected to. RT information is piggybacked on ordinary messages to minimize the network traffic.

5.3 Fault Tolerance

The DS has been designed to be as fault tolerant as possible, with the limited knowledge available. The Message Passing Layer assures reliable delivery as long as it is possible. Messages to a remote site that is in the Perm state will of course not be delivered, and can therefore be discarded. If a connection is lost, a reconnect will be attempted[1].

On top of this mechanism the protocols are implemented with recovery mechanisms, when possible. The mobile state protocol can bypass lost sites and avoid loss of the state. The Lock protocol, for instance, can recreate its mobile lock if it is lost.

5.4 Perm and Temp on the Internet

Detecting that a remote site is Perm gone on the Internet is hard, almost impossible. With dynamic addressing schemas as DHCP and and NAT's nothing can be said about the correct state of a machine. Perm is most likely detected on local LANs where distributed programs running over WANs will probably just experience Temp faults.

[1] These attempts will continue until a Perm is detected, but the interval between reconnection attempts will grow to avoid harassing servers.

6

Monitoring Tools

To help the programmer understand, tune, and debug distributed applications Mozart provides three tools for understanding the behavior of the DS.

- For real-time monitoring of communication, there is a panel tool called Distribution Panel that displays the current activity of the running site.
- To evaluate the result of a distributed application, the DS can be instrumented to write all events to a file. With help of a tool that displays logs graphically, logs from multiple sites can be displayed together in a graph.
- To simulate the behavior of a distributed application there is a tool that can simulate a set of Mozart sites in a controlled environment. The simulator executes Mozart code and simulates distributed events by a discrete event simulation engine.

6.1 The Distribution Panel

The Distribution Panel periodically retrieves the internal state of the Distribution Subsystem. The information is then displayed in a graphical tool (see Figure 6.1 and *"Distribution Panel"*). The Distribution Panel enables monitoring the behavior of a Mozart site in runtime. Information of currently known remote sites, exported entities, imported entities, sent messages summarized by message type, and marshaling information is displayed.

6.2 The Oz Message Tracer

The tool merges logs from different Mozart engines doing distributed computation. The logs are sorted in the casual order imposed by their internal message sending and receiving, and displayed graphically (see Figure 6.2). From the visualization of the casual order of events in a distributed computation the behavior of a set of Mozart sites can be examined in detail. The OzMessageTracer is available via Mogul[1].

[1] `http://www.mozart-oz.org/mogul/info/klintskog/ozmessagetrace.html`

Figure 6.1 An example of the Distribution Panel.

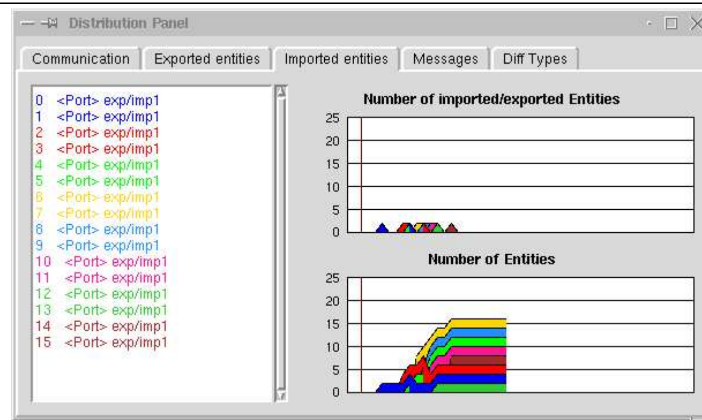
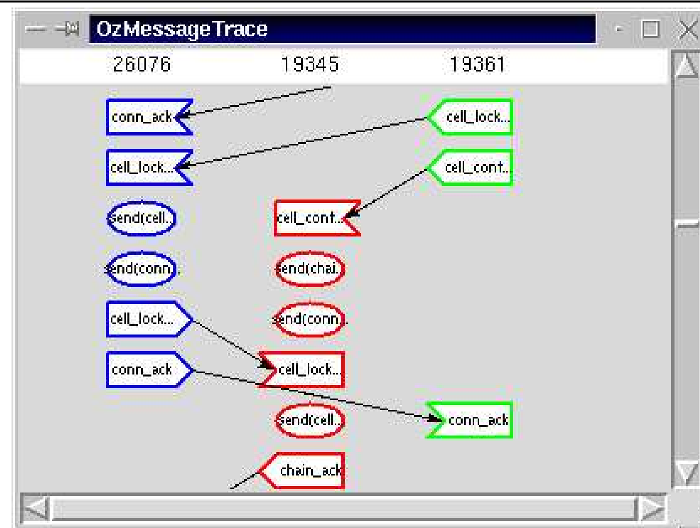


Figure 6.2 An example of the LogTool.

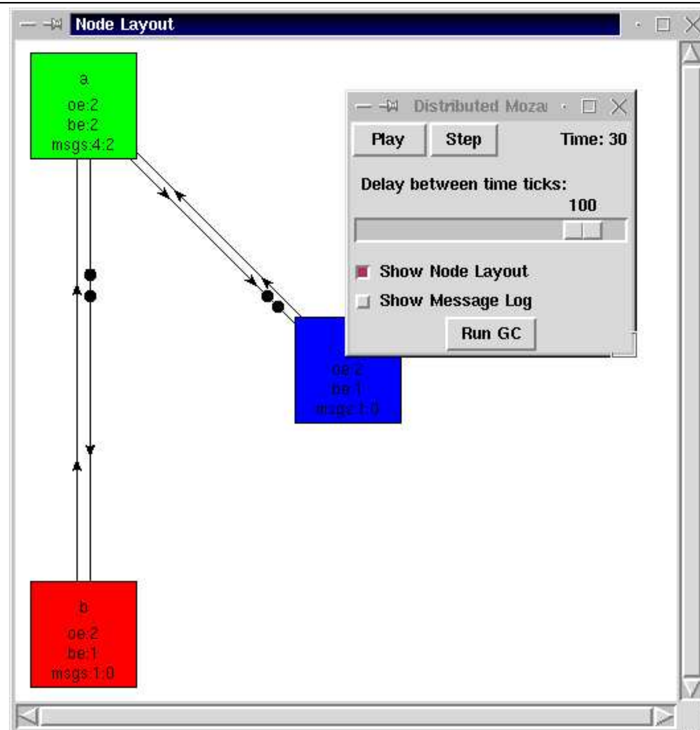


6.3 The Mozart Distribution-Subsystem Simulator

The simulator is a graphical tool (see Figure 6.3) for development and understanding of networked programming languages. It enables simulation of a set of communicating virtual machines in a controlled environment.

By giving the developer a tool where execution can be controlled, statistics can be retrieved and faults can be inserted, the simulator simplifies verification of new features for a networked programming language.

Figure 6.3 An example of the Simulator.



7

Interfaces

The Distribution Subsystem offers interfaces for both getting data and setting parameters to the Language level. The following modules interface the DS:

- DPInit[1]: Initialization of the DS.
- DPStatistics[2]: Various statistics on the performance of the DS.
- Remote[3]: Spawning remote processes.
- Fault[4]: Handling failure in customized ways.
- Connection[5]: Creating tickets and taking for bootstrapping connections.

There is also a set of properties that affect the behavior of the DS. These settings affect all distributed activities for the whole site so they should be used with care. They are defined in Chapter *Emulator Properties:* `Property`, *(System Modules)* under dp and dpLog.

[1]Chapter *Initializing and instrumenting the distribution layer:* `DPInit`, *(System Modules)*
[2]Chapter *Retriving statistical information from the Distribution layer:* `DPStatistics`, *(System Modules)*
[3]Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*
[4]Chapter *Detecting and Handling Distribution Problems:* `Fault`, *(System Modules)*
[5]Chapter *Connecting Computations:* `Connection`, *(System Modules)*