Distributed Execution of Ptolemy Models

Yang Zhao and Thomas H. Feng

EECS 290N Report
November 18, 2004

University of California at Berkeley
Berkeley, CA, 94720, USA

{ellen_zh,tfeng}@eecs.berkeley.edu

Abstract

In this paper, we offer an overview of distributed model execution with Ptolemy. Our discussion focuses on the Process Network domain, while the platform that we propose is domain-independent. Components such as DNS and PtAgents are introduced as separate processes in addition to Ptolemy with models running in it. They provide services important to the execution of the system, such as dynamic name lookup, data relay, and life-time measurement.

1 Introduction

Distributed modeling and simulation are an active research area. It has received increasing concern, particularly because high-speed networks and cluster computer systems become widely available. There are many good reasons for using such distributed systems in model execution, some of which are:

- to make full use of the combined power of the computing resources available in a network by involving them in a simulation, which is too slow or too large to fit into a single affordable machine;
- to simulate a real application featuring the ability of high-speed distributed computation;
- to build distributed collaborative shared data spaces; and
- to model the behavior of a P2P (peer-to-peer) or CS (client-server) network, to analyze its performance with a simulation identical to its real behavior, and finally to optimize it and re-engineer it.

Ptolemy II is a modeling, design, and simulation framework suitable for concurrent systems [1]. It supports a component-based construction of systems and it uses well-defined models of computation to govern the interaction between components. To leverage this framework to provide a formal specification and run time environment for distributed systems, and to incorporate the ability of high-speed distributed computation into Ptolemy II, we develop a platform for handling distributed deployment, remote interaction, and life-cycle management. We also explore how to integrate it with Ptolemy II cleanly (without substantial modification to the current framework) to execute a distributed system obeying some model of computation. We begin by defining the problem and requirements in section 2. We then discuss the platform design in section 3. Section 4 addresses some implementation issues. Section 5 concludes the paper.

2 Problem Statement

The goal of this project is to extend the Ptolemy framework to support distributed computation without requiring the user to know much techniques of distributed computation or to build up the low level communication mechanism from scratch. Assume a user has designed a system as a Ptolemy model and wants to distributedly execute it on multiple machines. The problem we address here is how to distribute the model transparently without requiring the user to modify the design, how to automatically deploy the system according to specific configurations, how to set up the remote communication mechanism, how to manage the execution life-cycle, and how to provide monitoring support for the user to monitor and debug the system.

We assume there are a set of computers in a network that are available for executing Ptolemy models. This set of computers may change over time due to computers connecting and disconnecting. In each of these computers, a entity called PtAgent is running, which is used for model distribution and remote communication. There is a global domain name service (DNS), and each PtAgent has the knowledge of locating it in order to find other PtAgents. Both the DNS and the PtAgents are implemented as separate processes interacting with each other with a network protocol, such as Java RMI, CORBA and HTTP.

A user can design a system as a Ptolemy model on any of these computers. A configuration specifies where to execute which actor, and whether an actor should be replicated to multiple computers. The configuration can either be manually fed in by the user or be determined by an algorithm based on some criteria, e.g. load balancing. The user is then able to press some button in Ptolemy and request to distribute the model and execute it automatically. We call the model for the whole system the *master model*. During the execution, the user may monitor the progress from the master model.

In order to support automatic deployment and execution, a set of tasks need to be addressed:

- *Discovery Mechanism* for each component to discover available resources.
- *Parser* to parse the master model and generate the submodels that will be deployed on the hosts according to the given configuration. This can be done by applying an XSLT transformer to the MoML file of the master model.
- *Launch Service* to launch a model on request and manage it's execution.
- *Remote Communication* for handling distributed interaction.
- *Monitor Service* for the user to monitor the execution.
- *Fault Tolerance* to handle host failures (future work).

3 The Design

In this section, we will discuss two approaches we have tried. Approach I experiments with Java RMI and extends the current Ptolemy infrastructure to mainly explore how to realize communication between distributed Ptolemy models. Approach II develops the discovery mechanism, launch service, monitor service and refines the remote communication mechanism.

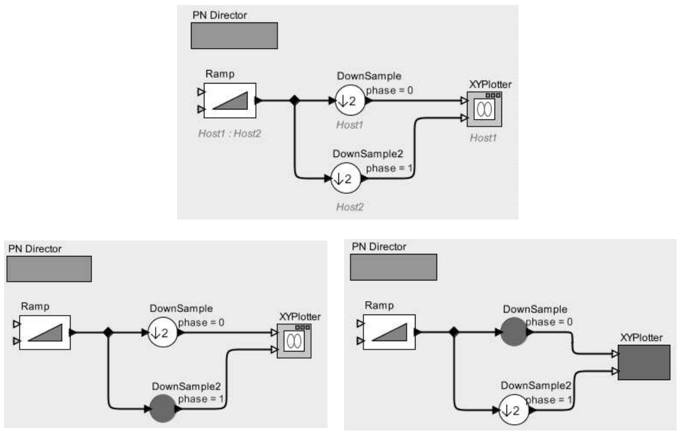

Figure 1. An example

3.1 Approach I

In this approach, we use a producer/consumer style of distributed communication for sending tokens to a remote process. We will focus on developing the infrastructure that supports remote communication between Ptolemy models. For this purpose, we assume we know which machine is available and we manually modify the master model's MoML file to generate the sub-models according to a configuration.

We choose Kahn Process Network (PN) as our interested model of computation for a set of distributed applications. Figure 1 shows a simple PN model and the two sub-models on `Host1` and `Host2`. This example may be too simple to make sense for distributed execution, but the concept we want to highlight here is the distributed execution conforming to the PN semantics. The configuration for the `Ramp` actor is shown as `Host1 : Host2` which means it will be distributed to both machines. We support such kind of replication since the `Ramp` data source is deterministic, and not computation intensive, the user may want to distribute it to both `Host1` and `Host2` so that no communication between it and the two down sample processes needs to be carried over the network. The gray icons represent processes running on a remote machine. The component representing the remote process is instantiated as a special actor: `DummyActor`. As its name suggests, this actor carries no computation, i.e. the actual computation is executed only on the machine that hosts the component. It is used here to represent the model's topology. Besides this, the `DummyActor` is also used to delegating data tokens received on its receiver to the corresponding remote process. In order to send the token to the remote process's receiver, we create a distributed object called `DistributedReceiver`. Here we chose Java

RMI as our underlying middleware for its implementation. The DummyActor's receiver works as a producer and the DistributedReceiver on the remote process works as a consumer. When ever there is token received on the DummyActor's receiver, the token is forwarded to the DistributedReceiver via a remote method call. The DummyActor has a special receiver, called `DelegatingReceiver` for handling the remote method call. To enforce the PN semantics on this model, we implement blocking read and blocking write between participants of message send and message receive over the network. The blocking on the receiver is realistically reflected on the sender side, as if they were executed on the same machine.

One difficulty we have with this kind of point-to-point distributed communication is to handle the starting sequencing. Note that `Host1` has some `DelegatingReceiver` needed to bind to some `DistributedReceiver` on `Host2` and vice versa. To find a distributed object, it needs to be created and registered with the naming service first. So we need take care of when to register a `DistributedReceiver` and when to bind to a `DistributedReceiver`. We first assume the `DistributedReceiver` is created during the pre-initialization phase and delay the binding until a `DelegatingReceiver` receives the first token (the earliest time could happen is in initialize phase). This also requires synchronizing the demons to start the execution after all of them have done the pre-initialization phase.

These experiments help us understand the problems and requirements for a distributed execution environment. However, it has some flaws. The strict start sequencing requirement reduces the flexibility and it could contribute to a big part of distributed exceptions. In the next approach, we will discuss a more flexible architecture that breaks down the dependency nicely and also facilities the discovery service and monitoring service.

3.2 Approach II

Figure 2 illustrates the main components in the PtAgent, and how they interact with Ptolemy sub-models. We assume each machine will run exactly one PtAgent process. There are four main module in the PtAgent:

- *Launcher* is responsible for distributing, loading, starting, and stopping a sub-model.
- *Discovery* provides the functions for Ptolemy to discover available computers or resources.
- *Relay* provides service for Ptolemy to transfer data to the sub-models located on other machines.

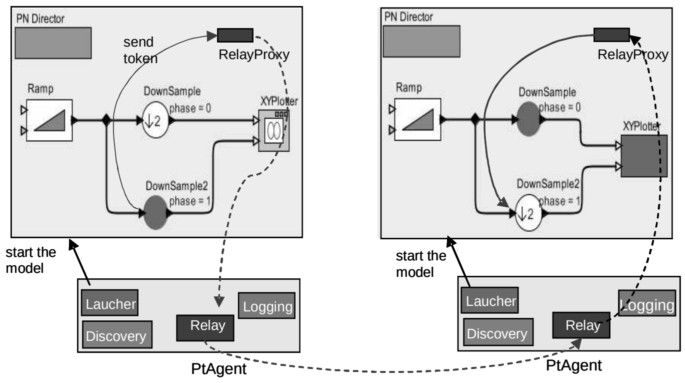

Figure 2. Interaction between 2 sub-systems

- `Logging` records the models' execution information and can be queried for debugging and monitoring purposes.

Each PtAgent may register itself to the DNS so that other PtAgents can find it. When a user wants to distribute a model (the master model), the `DistributeHelper` class will parse the model to sub-models and contact its local PtAgent (the only PtAgent running on the same machine) to distribute the model, which will in turn contact the related PtAgents to load and start the sub-models.

After a sub-model is started, it may need to transfer data to a remote sub-model, As shown in Figure 2, the `DelegatingReceiver` will contact an entity in Ptolemy called `RelayProxy` to send the data, which asks the PtAgent to relay the data to the corresponding remote PtAgent. Then the data is sent to the right port via the `RelayProxy` in the remote sub-model. This looks a bit complex, but there are several reasons to have it work this way. First, using PtAgents to relay data rather than sending the data directly to the remote receiver helps to break down the dependency loop as pointed in approach I. The idea is quite similar to CORBA using Event Channels to decouple the communication between a Producer and Consumer [2], except that here we distribute the Event Channels also, according to the configuration. Second, the `RelayProxy` is used to decouple the design of PtAgent and Ptolemy. Third, the `RelayProxy` can be domain specific to guard the execution semantics. For example, although it is possible to depend on some distributed middleware's blocking mechanism to throttle a "fast" process in PN, it is quite expensive. The `RelayProxy` can then be used to block the process locally. Fourth, the agent may record the data sending via it for debugging and monitoring.

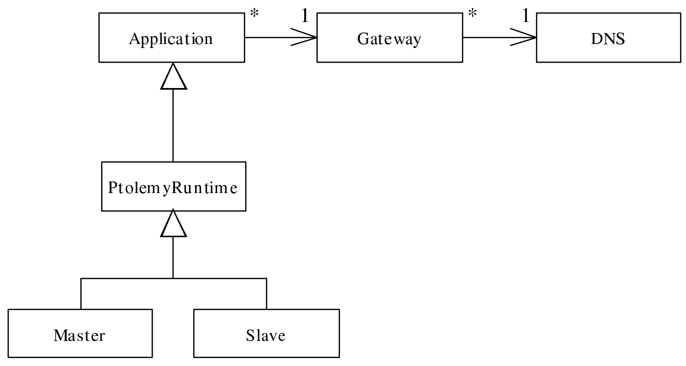

Figure 3. Class View

4 Implementation

We classify the three types of processes in our system as DNS, PtAgents and applications such as Ptolemy with (sub-)models running in it (Figure 3).

The DNS can be regarded as a repository to look up available resources in a given domain. It is assumed that there is only one single DNS. All other components have the knowledge of locating the DNS initially, so as to use resources in the domain or to be used as a resource. Criteria are used to query the DNS to locate other components in the system. We have designed several kinds of criteria, such as name query, interface query and behavioral keyword query. For example, a sub-model may issue a name query with a regular expression. The query is sent to the DNS via the PtAgent on the same machine as the sub-model. The set of sub-models with names matching the regular expression is then returned. (It may be empty when no matching sub-model has been registered in the DNS.) This result also contains the information required to initiate connections to the PtAgents on the machines of those sub-models. Lookup criteria may be specified in the configuration, or they may also be generated by the execution of the sub-model. This makes the lookup mechanism very flexible.

A PtAgent acts as a startup agent, a relay, and a firewall for Ptolemy model(s). We restrict that any application can only communicate through a PtAgent for the reasons stated in Section 3.2. To register itself to the DNS or to look up other sub-models, Ptolemy may not directly contact DNS but it sends commands to the PtAgent. Similarly, messages sent between two sub-models must cross two PtAgents, the one working with the sender and the one on the receiver side. Instead of merely relaying commands or messages, PtAgents also provide useful functions for the startup and maintenance of the distributed system. One important function is life-time management. By periodically sending an "`isAlive`" message to Ptolemy and waiting for its response, the PtAgent timely detects the malfunction of Ptolemy processes. Another example of such functions is authentication checking every time when a sub-model is being connected to.

5 Conclusion

We have discussed the high-level design of a distributed platform for Ptolemy II. This platform enables distributed model execution without requiring much experience of the user. The use of a global name system and a PtAgent for each machine greatly improves the flexibility of a dynamically deployed system.

We limit our current implementation to the PN domain, but we are also careful enough to make our platform easily extensible to other domains supported by Ptolemy. For distributed execution of timed domains, fault tolerance and backtracking techniques are required. These issues will be studied extensively in our future work.

References

[1] S. S. Bhattacharyya, E. Cheong, J. D. II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java volume 1: Introduction to ptolemy II. Technical report, July 2003.

[2] C. O'Ryan and D. L. Levine. Applying a scalable CORBA events service to large-scale distributed interactive simulations. In *Proceedings of the Fifth International Workshop on Object-Oriented Real-Time Dependable Systems*, 1999.