

Distributed Application Framework

A Distributed Application Framework for TLUMIP

Introduction

The TLUMIP model development process has reached a crossroads, namely the computing time required to run a forecast year exceeds the time that we are willing to wait! This crossroads presents several options:

1. Simplify the models to run faster.
2. Buy bigger, faster hardware.
3. Distribute the problem over more hardware.

We should proceed with the basic assumption that models as developed represent the simplest models which will provide meaningful forecast data. Otherwise, we would not be at this crossroads! Some of the TLUMIP modules are multithreaded and can take advantage of multiple processors in a single box so buying bigger (more CPUs) hardware has an immediate appeal. Unfortunately, the cost increases almost exponentially as processors are added to a box. More importantly, the upper bound on the number of CPUs required in a box has not been determined. Should we buy a 16, 32, or 64 CPU system?

Given these constraints, the reasonable path forward is to distribute the TLUMIP model over a number of boxes. This path provides a number of benefits:

1. Use commodity hardware for each "node" in a distributed framework.
2. The number of nodes can be easily changed over time.
3. The "nodes" can be affordably upgraded as hardware advancements become available.

The real question is, "Can the TLUMIP modules be distributed such that they provide sufficient concurrency to make the effort worth while?" The honest answer at this time is "probably". A review of the Transportation Assignment module (TS) shows that multi-threading has helped to reduce run-time in an almost linear fashion. Thus, the TS module is a prime candidate to be distributed over more hardware.

Distributed Frameworks

There is a plethora of literature on parallel computing systems and distributed application frameworks in the computing field. The distinction between the two is important. Parallel systems tend to favor computing problems which are concurrent by nature and thus design these systems for pure concurrency and speed.

In contrast, distributed frameworks tend to be designed for problem which are not inherently concurrent. They support problems which may prefer to execute sequentially, provided they could execute with adequate speed. Key to distributed frameworks is collaboration. These frameworks deal with the collaboration among collections of people, information, and objects.

Another important concept in framework development is that of whitebox vs. blackbox. Whitebox frameworks require application developers to have intimate knowledge of the frameworks' internal structure. Although whitebox frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies. In contrast, blackbox frameworks are structured using object composition and delegation more than inheritance. As a result, blackbox frameworks are generally easier to use and extend than whitebox frameworks. However, blackbox frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use-cases.

Given the scope of the TLUMIP project, the use of a black box framework is preferred but this underscores the need to focus the development of the framework to provide the best return.

TLUMIP

The key to success in creating a distributed framework is to adhere to the requirements of a class of problems. While distributed frameworks share common principles, a good framework will target a class of computing problems and provide the appropriate level of infrastructure. In this sense, a distributed framework can be viewed as a concrete reification of families of design patterns that are targeted for a particular application-domain.

A lightweight distributed framework being proposed for TLUMIP. In short, the goal of this framework is to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment. The name underscores the goal of providing a distributed framework which is targeted for a particular application-domain, namely our domain! Meaning, if other people can use the framework, great. But, that's not one of the key requirements of the framework development.

Framework Components

The relationship between frameworks and components is highly synergistic, with neither subordinate to the other. Frameworks can be used to develop components, whereby the component interface provides a *Facade* for the internal class structure of the framework. Likewise, components can be used as pluggable strategies in blackbox frameworks. In general, frameworks are often used to simplify the development of infrastructure and middleware software, whereas components are often used to simplify the development of end-user application software. Naturally, components are also effective for developing infrastructure and middleware, as well.

The key components in this framework are:

1. Tasks - These are units of work which developers create. A task will generally be executed by a thread.
2. Ports - These are the means by which tasks communicate. Ports can be used to send messages or data to other tasks.
3. Messages - Represent some event that needs to be sent from one task to another. Messages also represent how a task communicates with the framework.

4. Framework components - A number of components will be made available to a task to reduce the work a developer has to do.

Using these simple constructs, it's possible for a variety of tasks running over a set of distributed machines to execute independently but communicate when required. Following the black box approach, these components will hide the complexity of execution from the developer.

Tasks

A task represents an executable piece of work. In the Java world, this would be a *Runnable* component. By abstracting this to the task level, the framework can provide a thread pool from which tasks are run. The framework manages the details of managing threads.

Application will be built up from one or more tasks. A simple application may consist of a single task running on a single node. A sophisticated application may be built from many tasks running on several nodes.

Ports

A port represents a connection to another task. If the destination task is running on the same node or machine, this connection might be a pipe. If the destination task is running on a remote machine, the connection might be a TCP connection. The framework abstracts the communication details from the developer and chooses the most efficient connection type based on the locality and overall load in the system.

Central to the management of messages over ports is the Queue. Each port will have one or more queues which may hold both inbound and outbound messages. This allows tasks to operate asynchronously when sending a message. The message can be "sent" but the task does not have to wait until the message is sent and received by the destination task(s). Similarly, messages can arrive for a task and be queued until the task is ready to process them.

Ports and the underlying queues can be blocking or non-blocking. Meaning a task can wait for the arrival of a message or move on to other work. Ports can be shared between tasks.

Messages

A message represents an event or data that is sent from one task to another task. Messages can be of any type or size. Messages can be cached. This behavior is determined by the sender. In some cases, it might be useful to cache a message for later use.

Framework components

Framework components represent components which are available to all tasks. Fundamentally, these components provide the basic runtime environment for tasks. Framework components include environment information, logging, and communication.

In addition to these components, tasks may make use of any native Java library component.

Incremental Development

The distributed framework will be developed in increments. This approach avoids the risks behind the big bang development approach. Many of the complex features found in a distributed system can be deferred until the initial development is complete. Following this approach, only the core framework components will be developed.

Notable features that will be delayed until a later release include:

1. Dynamic node discovery. Nodes will be able to find other participating nodes using IP multicast. This feature can be deferred in place of static properties files.
2. Dynamic task execution. Tasks are bound to nodes at runtime. This feature can be deferred in place of static task loading. Meaning, tasks can be bound to nodes at deployment time.

Many of the core components have been developed. Including:

- Thread Pool
- Queue
- Chained Exception
- CacheTable

These components were developed during the first phase of the TLUMIP model development project.

Industry Standards

Every effort will be made to make use of existing components and industry standards. For example, there will be an administration console for the distributed framework to view that status of running tasks on each node. This console will be web based and use an imbedded version of the Tomcat web server. This is an open source JSP/Servlet Engine provided from Apache.