TOWARD AN INTENSIONAL MODEL FOR PROGRAMMING LARGE SCALE DISTRIBUTED SYSTEMS

W. DU
Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. Canada E3B 2S6
E-mail: wdu@unb.ca

This paper introduces a distributed programming model based on intensional programming paradigm. The model provides constructs to specify collaborations of distributed processes through their communication patterns. It models a distributed system consisting of nodes to execute distributed processes as an intensional context space, and models communications between processes as intensional context switching operations. For each communication pattern among distributed processes, an intensional dimension can be defined by context switching methods. The intensional semantics of the context switching methods of a dimension defines the corresponding communication pattern. In programming, collaborations of distributed processes can be expressed by defining a context space modeled from the underlying system and a set of dimensions for inter-process communications. The paper also describes a Java implementation of the model, a Java API called JavaTopology.

1 Introduction

The rapid growth of Internet-based large scale applications such as WWW brings new directions for distributed computing and raises complexity of such distributed systems in a new level\textsuperscript{12}. Scale, variety, and complexity of communications are important characteristics of these new waves of distributed systems. These systems usually involve hundreds and thousands participating nodes in an Internet-based distributed environment. Distributed processes on these nodes collaborate each other in some way through inter-communications. The contents transferred during communications between distributed processes varies, such as data, objects, codes, and multimedia documentation. Collaborations of distributed processes in such large scale distributed systems can be very complex, considering their sizes and distributed tasks.

We can classify large scale distributed systems into three categories in terms of their communication complexity. In a simple multi-client/single-server system, the communication pattern and content of communication between a client and the server are fixed. For example, the early version of WWW is such a system. It has very simple star-shape communication pattern. Each client can send a request to the server to retrieve a Web page on
the server. There is neither collaboration between clients, nor collaboration between servers to handle a client’s request. The content of communication from a client to a server is a simple request with URL, and that from the server to the client to response the request is an HTML file. The main difficulty in designing this kind of systems is focused on the behaviors of client software, i.e. browser, to generate requests and handle received data, and server side software to handle possibly large amount of requests from clients. The client and server software is permanently deployed on the client and server side, respectively. In terms of complexity of distributed algorithms, however, such kind of systems are simple.

In a flexible multi-client/single-server system, the contents of communication between a client and the server can not only include static data but also involves migration of computing tasks from server to client or vice versa. For example, WWW with Java support is such a system. A Java applet can migrate from the server to a client and let the client to compute the task programmed in the applet. In such distributed system, instructions of a computing task can be stored in one node, dynamically migrated to another node and be executed on that node. There is even more complexity involved when data is considered. For a migrating computing task, it must specify where its input data come from, from the destination node and/or the source node. Also the forms of code migration can vary. For example, in the current technology, code can be migrated as an applet to a browser or as an argument or result object of a remote method invocation. These all increase the complexity of distributed programming in order to deal with all those programming aspects, though the communication pattern itself still simple only involving two (client/server) nodes.

The most complexity comes from flexible multi-server systems. Such a system involves multiple collaborating servers in a large scale distributed environment. Here we consider that a client which allows others to collaborate with is both a client and a server. In a large scale system, possibly many servers work together with very complex collaboration and communication patterns to accomplish an end user’s request. The distributed multi-server systems for WWW and SUN’s Jini system are two examples. Computations involved in these systems usually based on complex distributed algorithms. These distributed algorithms have to not only deal with traditional distributed computing problems, but also face new issues, such as the system size, static and dynamic deployment of system resources, dynamic migration of computing tasks and resources, and collaboration patterns with dynamic resources. If the scale of a system is small, i.e. it only involves few collaborating servers, the programming task is relatively simple. For each server can be programmed
individually. However, when the system scale is large or very large involving tens and hundreds collaborating servers, programming individual servers become impossible. There must be a way to specify a distributed algorithm for the entire system or all the servers and to implement the algorithm in a single distributed program or automatically generate distributed programs for individual servers according to a single specification.

For example, a sprinkler system of a large city can have a computer to control each park’s sprinkling work. There are possibly more than one hundred parks in the city. To keep the city’s water pressure, sprinkling has to be done park by park. The completion of sprinkling in a park is detected by a sensor in the park’s lawn. For such system, a distributed algorithm has to be designed and a program written or generated from the algorithm has to be deployed to each computer statically or dynamically. In terms of programming, the challenging work here is to specify the distributed algorithm in a single specification and generate its implementation(s).

Current technologies give great support for peer-to-peer and server-centric collaborations in distributed applications. These technologies can be classified into two categories, message passing and shared space. In message passing, collaborations of two distributed processes on different nodes can be implemented by exchange messages between the two. There are two support levels for message passing based collaborations. At network level, basic networking facilities are provided, such as socket, for programmers to build message passing by themselves. At the middleware level, there are pre-built message passing-based communication infrastructures, typically implemented by remote method invocations of distributed objects, such as CORBA and Java RMI. In shared space technology, collaboration between two distributed processes can be specified by letting the two to access a shared space on a server. Thus collaboration can be built by high-level synchronization mechanism. JavaSpace built on Jini is an example of such technology. These technologies support client/server-based distributed systems well. For communication patterns of such systems are basically star-shape, i.e. multiple clients talk to a single server. For example, in designing our sprinkler system, we can program the computer at each park to communicate with a central server to synchronize its sprinkling work with others. These technologies also support client/server and server/server collaborations with implicit synchronization mechanism through naming or look-up services, such as those in CORBA and Jini. This mechanism gives great support to certain distributed applications. For example, using a Jini system, we can let a single Sprinkling service join the system and let multiple clients, namely computers in parks, to look up the service. The service can be assigned to a client exclusively at a time by the
server. However, this implicit synchronization mechanism does not support explicit specification of communication patterns, especially in terms of the topology of communications among clients and servers. For example, in the sprinkling system, we may want to specify the topology of the park computers in terms of their neighborhood relationship which the order of their sprinkling work depends on.

In the following sections, we propose a new distributed programming model that supports explicit specification of collaboration patterns among distributed processes in terms of their communication patterns. Our objective is to complement other distributed programming technologies, hence give distributed application programmers the needed expressive power in a simple and practical approach. In Section 2, we introduce the proposed distributed programming model which is based on the intensional programming paradigm. In Section 3, we discuss implementations of the proposed model in programming languages and application development tools. In Section 5, we give some concluding remarks.

2 The intensional distributed programming model

The intensional programming paradigm has great potential to be the basis of a successful distributed programming model. In the intensional programming paradigm, we write a single program whose meaning depends on an implicit context space. In an intensional program, an operations in one context can depend on operations or values in other contexts. This kind of dependencies can be expressed using context switching or intensional operators. In other words, at program level, an intensional program can have different behaviors depending on the underlying context. Also, at individual operation level within the program, an operation can be performed in a context different from the underlying context, or an operation in the underlying context can depend on other operation(s) performed in other context(s). Here by operation we mean a simple or complex computing task. As discussed in the previous section, the proposed distributed programming model is to specify, using a single specification, a distributed system’s overall communication topology that consists of a set of independent or inter-related communication patterns. This specification can precisely specify collaborations of distributed processes and their individual behaviors which are to be performed potentially on tens and hundreds of nodes.

The past experience in research on intensional parallel programming give us many helps in building the intensional distributed programming model. In intensional parallel programming, there is context parallelism. That is, a
computing task in a context can be executed in parallel with computing tasks in other contexts, unless it depends on some computing tasks in other contexts. In intensional parallel programming, we write a single program to specify parallel computing tasks and their dependency relationships in all contexts of the underlying context space. We try to exploit as much context parallelism as possible in the program. We use intensional operators to explicitly specify the inter-context relationships of parallel tasks in the program. That is, whenever an intensional operation is specified, there always be a dependency between computing tasks in two contexts. In terms of implementation of intensional parallel programs, especially in distributed memory or multicore environments, we can always schedule parallel tasks resulting from context parallelism onto different processors to achieve true parallelism.

In designing the intensional distributed programming model, we can use a similar idea of context space and context switching to that in intensional parallel programming. However, the two have different emphases and have to deal with different issues. In distributed programming, we do not focus on exploiting parallelism. Parallelism may come natural in a distributed algorithm, since potentially distributed processes can be executed in parallel on different nodes. Instead, we focus on specifying collaborations of distributed processes precisely and concisely in terms of their communication patterns. In other words, in intensional parallel programming, the less intensional operations are used, the more parallelism can be exploited. In contrast, in intensional distributed programming, we can use explicit intensional operations to specify communication patterns of distributed processes.

In designing the intensional distributed programming model, first we have to decide what should be the context space that the entire programming model is based on. The context space can be dynamic but must be stable. That is, we should allow new contexts being added to the space and existing contexts being removed from the space. However, all contexts in the context space must have stable identities that can not be changed dynamically. Otherwise, it is impossible to design meaningful context switching operations. Based on this argument, in a distributed system, we must exclude those migrating entities such as code, objects, data, documents and other resources as potential contexts. For their possibly frequent migration may result in easily losing count on their identities in programming. The only entities in a distributed system that can not migrate is the participating nodes. Here by node, we mean a computing host on which a distributed process can run. Practically a node can be identified by a URL. Thus the context space in the intensional distributed programming model can consist of a set of nodes represented by their URLs.
In the following we informally describe the model.

1. A context space consists of a set of identifiers each of which can identify a participating node of the underlying distributed system.

2. Using user-defined `addContext` method, a new context can be added to the context space dynamically. Using user-defined `removeContext` method, an existing context can be removed from the context space dynamically.

3. A context relationship, called `dimension`, in the context space can be defined through user-defined context switching method(s). For example, to define a uni-directional ring relationship or dimension, a `next` method can be defined, and its semantics should guarantee that each context has a unique next context and all contexts in the context space are connected. The method should also handle the case when contexts are dynamically added or removed.

   Multiple dimensions can be defined. They can be orthogonal or related. Two dimensions are related if in one’s context switching method definition, the other’s context switching method is called.

   Parameters and result types of a context switching method are also user-defined depending on the underlying application. In other words, a context switching method does not have be just for defining the context relationship. It can do more according to the application, but the context relationship must be `derivable` from the method.

4. In distributed programming, a dimension can correspond to a communication pattern among nodes that are represented by contexts.

   Typically a dimension can correspond to a migration pattern of certain entities in a distributed system. For example, we can define `CodeDimension`, `DocumentDimension`, and `ObjectDimension`. Each dimension with its context switching method(s) specifies a unique communication or migration pattern of the corresponding entities among the nodes in the system.

5. A distributed program designed using this model has intensional semantics in terms of its multidimensionally structured context space. This program defines the behavior of the distributed process on each node in a distributed system. Without using context switching methods explicitly, the program defines the computing task local to the node, i.e. local to the underlying context which is implicit. A call to a context switching method in the program always denotes a communication from the local
node to a remote node or context switching from the underlying context to another context. The semantics of this denotation for communication is implemented by the semantics of the context switching method defined in the underlying programming language.

For example, in our sprinkler system example, we can write a single distributed program for all computers that control sprinkling work for the city's parks. The collaboration pattern among the computers or nodes can be simple. We can organize the nodes in a uni-directional ring. Except for the starting node, all other nodes have to wait until their neighbors to complete sprinkling work. In this case the communication between the two neighbors is just a synchronization signal. Thus, we can define a \textit{Synchronization} dimension to specify this communication pattern. The dimension can have one context switching method \textit{next}. The \textit{next} method switches context from the underlying context to its neighbor in the ring. The \textit{next} method can be implemented in various ways depending on different implementation strategies of the system. The following are three possible strategies.

1. \textbf{URL next()} returns the URL of the neighboring node of the underlying one. In the program, we can write a remote method call using the URL returned from \textit{next()}. The following is an example pseudocode.

   \begin{verbatim}
   doSprinkling();
   signalNeighbor(Synchronization.next());
   \end{verbatim}

2. \textbf{void next()} \texttt{self} can be a remote method call to the neighbor. In this case, we can write

   \begin{verbatim}
   doSprinkling();
   Synchronization.next();
   \end{verbatim}

3. \textbf{void next(Object syncObj, Method syncMethod} can perform a remote method invocation on the neighboring node using the given remote object and method. In this case, we can write

   \begin{verbatim}
   doSprinkling();
   Object obj = getSyncObject();
   Method med = getSyncMethod(obj);
   Synchronization.next(obj, med);
   \end{verbatim}
In this sprinkler distributed system, we can also dynamically distribute the controlling code to the nodes, instead of statically deploying the code. In this case, we can define a Code dimension to specify the migration pattern of the code. For example, if the code comes from a single source node, we can define a context switching method, origin, for Code dimension that switches from any context to the source context or node. In this design, we can write

```
Class sprinkler = (Sprinkler) getCode(Code.origin());
```

Alternatively, the code can also be distributed in a ring pattern but the neighborhood relationship or the communication pattern can be different from that of Synchronization dimension. In this case, we can define a next method for Code dimension too, plus a isSource method.

```
if(!Code.isSource()) {
    Class Sprinkler = (Sprinkler) getCode(Code.next());
}
```

3 Implementation strategies

There can be various strategies for implementing the intensional distributed programming model, as long as the design goal of the model can be achieved. That is, based on intensional semantics, we should be able to write a single specification for multiple collaborating distributed processes. Here the specification itself can be the executable program performed by each process, or from it multiple programs can be generated for corresponding processes.

3.1 Programming style

Most loosely, we can just implement the model as a style of distributed programming. For each communication pattern in a distributed program, a dimension can be defined as a program structure such as a class. All methods to be used to realize a communication pattern using context switching, as well as the state related to the communication pattern, can be defined and encapsulated within the structure. Thus this dimension structure provides not only a single place for defining a communication pattern, but also a name space for referencing to or using it in the program.

3.2 Application Programming Interface

At programming language level, the intensional distributed programming model can be implemented as an API. The API can provide an application
framework for distributed programming based on the model, as well as founda-
dation classes or libraries to support this programming model.

We have designed a preliminary version of such API for Java called Ja-
vatopology. JavaTopology consists of three parts:

- A set of framework superclasses that provide a framework for implement-
ing the intensional distributed programming model.

- A set of topology manager or planner classes that define many stan-
dard context switching topologies and provide facilities for user-defined
  topologies.

- A set of communication manager classes that provide infrastructures for
  various implementation of context switching methods at remote commu-
nication level.

The following is a brief description of JavaTopology.

3.3 ISpace and IDimension classes

ISpace class implements the context space. (Here 'I' stands for "Inten-
sional".)

class ISpace {
    Vector contextSpace; // store contexts (URL objects)
    URL currentContext; // store the underlying context

    boolean isIn(URL url); // check if url in the space

    boolean isCurrentContext(URL url);
    // check if url is the current context

    void setCurrentContext(URL url);
    // set url as the current context

    URL getCurrentContext(); // get the current context

    void addIDimension(IDimension id);
    // add a dimension from the space

    // remove a dimension from the space
    void removeIDimension(IDimension id);
void addContext(URL url, IDimension[] ids);
   // add a new context into the space.
   // The IDimension array is optional.
   // the context will also be added to each given dimension.
   // Otherwise it will be added to all dimensions

void removeContext(URL url);
   // remove url from the context space and all dimensions

ISpace(URL[] initSpace, ICommunicater comm);
   // the both parameters are optional.
   // initSpace gives initial contexts to the space.
   // comm gives an implementation of context switching methods
   // for all dimensions

Adding and removing contexts at run time are dynamic, that is, we need to update the context spaces of all running programs in the system through a communication pattern. We implement addContext and removeContext methods by using JavaTopology itself with an internally defined dimension.

**IDimension** class is the superclass of all built-in and user-defined dimensions.

class IDimension {
    Vector subContextSpace;
       // store contexts of this dimension

    ICommunicator icommunicator;
       //implementation of context switching methods

    boolean isIn(URL url);
       // check if url in this dimension

    void addContext(URL url);
       // add a new context into this dimension
       // as well as the context space

    void removeContext(URL url);
       // remove url from this dimension

    ISpace(ISpace ispace, ICommunicator comm);
// both parameters are optional.
// ispace can be used to select a subspace for the dimension
// by default, the dimension takes all contexts in the ISpace
// comm gives an implementation of context
// switching methods for this dimension
}

The context relationship that defines a communication pattern in a dimension
does not have to involve all contexts in the context space. It may only involve
a subset of the context space. Also the context switching methods in a di-
mension can use a different communication implementation than the default
one bound to the ISpace.

3.4 Standard and user-defined IDimension subclasses

JavaTopology provides standard dimensions each of which, as a subclass of
IDimension class, implements a standard communication pattern topology,
such as ring, mesh, cube, and hypercube. A standard dimension is config-
urable to set up the topology in different ways. A standard dimension can be
used as the basis or superclass of a user-defined IDimension class that uses
that standard topology. We define one or more context switching method in a
standard dimension to switch context from the current context to another one
in the dimension. A context switching method in a standard dimension always
has type URL methodName(). For example, the following is the definition of
IRingBase class that implements a bidirectional ring topology.

class IRingBase extends IDimension {
    final static int RANDOM, ISORT, DSORT,
    FIF /*First In First*/, LIF /*Last In First*/;
    URL left(); // returns URL of left neighbor in the ring
    URL right(); // returns URL of right neighbor in the ring

    IRingBase(ISpace ispace, ICommunicator comm);
    IRingBase(ISpace ispace, ICommunicator comm,
        int configurationParameter);
}

The ring topology among contexts in a dimension can be set up in different
ways. By default, the neighborhood relation of a context is set randomly.
It can also be set in increasing or decreasing order of their IP addresses, or
according to the time at which the contexts are added to the context space.

The following is a user defined dimension extended from the IRingBase
class for our sprinkler system example.
class SprinklerRing extends IRingBase {
    URL starter;
    void setStarter(URL url);
    // set the starter in the dimension

    boolean isStarter();
    // check is this is the starter

    void signalNeighbor() {
        doSprinkling();
        icommunicator.communicate(right());
    }
}

void doSprinkling();

SprinklerRing(ISpace ispace, ICommunicator comm);
SprinklerRing(ISpace ispace, ICommunicator comm,
              int configurationParameter);
}

For non-standard communication topologies, user-defined IDimension classes can be defined by subclassing IDimension class. To reuse a user-defined topology, a base IDimension class should be defined in a similar way to the standard ones. In the class, any topology layout algorithm can be used to set up a user-defined topology by defining corresponding context switching method(s).

3.5 ICommunicator interface and implementation classes

An ICommunicator object is in charge of communication between two nodes using a user selected infrastructure. A context switching method of an IDimension is implemented with an ICommunicator bound to the dimension. The ICommunicator interface defines a common interface for all ICommunicator implementations. The interface has only one method:

interface ICommunicator {
    Object communicate(URL url, Class cls, Method med,
                       Object[] parameters)
}

Except URL parameter, all other parameters are optional. This method invokes the given remote method of the given class at the given URL with the given parameters. It returns null or an object returned from the remote
method invocation. By default, the given class and method are the current
class and method in which the communicate method is called.

In JavaTopology, we implement the ICommunicator interface with the
following classes.

**JavaTopologyICommunicator** is the default ICommunicator if no other
ICommunicator is explicitly bound to the context space or a dimension.
It does not depend on other distributed infrastructures. When a Ja-
vTopologyICommunicator is created from this class, it behaves like a
server to communicate with other JavaTopologyICommunicators running
on other nodes.

**JiniICommunicator** is a Jini implementation. To run the program, it needs
Jini’s look-up and join services.

**RmiICommunicator** is a Java RMI implementation. To run the program,
it needs RMI’s naming service.

**CorbaICommunicator** is CORBA implementation. To run the program,
it needs an underlying ORB on each node.

### 3.6 An example

The following is a simplified JavaTopology program for our sprinkler dis-
tributed system using the above defined SprinklerRing class in JavaTopology.

```java
import JavaTopology.*;
class Sprinkler {
    public static void main(String[] args) {
        // get this node's URL from command line
        URL myURL = new URL(args[0]);

        // create an ICommunicator using JINI implementation
        ICommunicator myIComm = new JiniICommunicator();

        // create an empty ISpace and set default ICommunicator
        ISpace myISpace = new ISpace(myIComm);

        // create an IDimension
        SprinklerRing myIDim = new SprinklerRing();

        // add this context to the IDimension and ISpace
        myIDim.addContext(myURL);
    }
}
```


4 Concluding remarks

In this paper, we report the preliminary result of our research on the intensional distributed programming. This research has two objectives. First, we attempt to build a new model that supports distributed programming with explicit specification of collaborations of distributed processes through communication patterns or topologies. The intensional distributed programming model is distinct with other distributed programming models in several aspects. It is a high-level programming model. The model is independent of low-level implementations. Implementations of the model in terms of context space and context switching can be abstract and independent of networking and middleware technologies. It provides a different distributed programming paradigm. That is, it programs a distributed system as a whole, and builds the system by specifying collaborations of distributed processes in high-level constructs, instead of programming individual processes. Also, it gives the programmer a different approach to writing distributed programs using the notion of context space and context switching. Finally, all the above distinct features of the model are built on the model’s formal intensional semantics.

Secondly, we attempt to develop a new application area for the intensional programming paradigm. This and all past research on intensional programming has shown that intensional programming is one of the best programming paradigms to support programming those software systems that involve large
scale entities with regular collaboration patterns. Examples of such systems include mathematical and numerical system with multidimensionality, massively parallel systems, and Internet-based computing systems. The paradigm provides a logical and concise way to program such large software systems with sound semantics. Compared to other intensional systems, the intensional distributed programming model described in this paper shows a new experience in developing intensional systems. That is, an intensional system does not have to be a self contained system; it can be built on top of other technologies and their associated semantics.

Acknowledgments

This research is supported in part by a Canadian NSERC grant.

References