Composition of Bioinformatics Model Federations using Communication Aspects

Keith L. Lee, David Stotts
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
{lee,stotts}@cs.unc.edu

***Abstract*—Scientists in bioinformatics research utilize multiple software tools and models for their analysis of genomic data. This involves a mixture of stand-alone and custom software. Stand-alone application tasks include computational data analysis and data visualization. The genomic data links the software together; the output from one may be suitable as input for another. Scientists often create custom tools and scripts to perform additional computational and processing tasks, such as filtering, before passing data from one tool to the next. This interoperating collection of tools and models becomes a "federation" of collaborative software. We propose an approach using aspect-oriented programming and Linda-style tuple space coordination language to facilitate connecting the individual components of the federation together; we assemble the separate components into the larger complex system, ideally without altering the original components. This paper describes the concepts behind our approach of using AOP and tuple spaces to intercept, filter and transform data as well as manage tool execution and coordination.**

Keywords-Aspect-oriented Programming, Modularity, Software Federations, Tuple Spaces

I. INTRODUCTION

Many modern scientific problems require domain knowledge beyond that of any one scientist. Scientists across and within the spectrum of disciplines collaborate to work on shared research interests. Inter- and intra-disciplinary teams combine their expertise; each one develops software, tools and models using their domain knowledge to solve a smaller piece of the larger problem. For scientists to work together, their software, models and tools must also work together.

The individual pieces of software chain together to create larger more complex interoperating software systems termed "federations". The results of the individual computations and calculations often manifest as output data and data visualizations. Output of one application becomes the input for another. This exchange of shared data binds applications together, but this coupling is not necessarily straightforward.

Researchers working together may not have a unified programming environment. Differences can range from programming languages to computing platforms to data representations; the list goes on. Many programs function independently, solve separate scientific problems, or serve broader computational purposes. A bioinformatics tool chain may have programs specific to a particular biological science problem (SNP genotyping) combined with generalized biological science analysis applications (sequence extraction and manipulation) and generic data analysis (histogram).

When researchers combine software to work together, data mismatches arise. They could be spatial, such as trying to match geographical locations. They could be temporal; a program recording data hourly exchanges data with one expecting daily average measurements. They could even be mathematical, such as significant digits. For the software federation to work, mismatch differences must be resolved.

Adding mismatch manipulation code directly to one tool couples it to the other. Updating one with a new technique, data format or tool means updating the other to match. Coordination and communication are also problems. The output of tool A becomes the input for tool B, tool C must communicate state and data with tool D, etc. Placing this logic in a separate bridge reduces coupling between the tools.

In earlier work, aspect-oriented programming handled bridging data mismatches as a mismatch manipulation module. Here we expand our approach by combining it with tuple spaces to form coordination and communication modules. We treat communication, and the exchange of data between components, as problems separate from component functionality. We aim to connect the separate programs into a scientific model federation, ideally without altering the source code of the individual components of the federation.

This paper presents our continued work on scientific model federations. We use multiple, commonly used, freely available stand-alone and web-based software applications in conjunction with custom software. These applications use a variety of programming languages. We use aspect-oriented programming plus Linda-style tuple spaces to code the communications aspects.

II. BACKGROUND PROGRAMMING PARADIGMS

A. Aspect-Oriented Programming

The aspect-oriented programming (AOP) [1, 2] paradigm uses separation of cross-cutting concerns to improve software modularity. Traditional computer programming paradigms, such as object-oriented programming, procedural programming and functional programming, modularly separate and encapsulate various parts of program hierarchy. Aspect-oriented programming augments them by separating and encapsulating structures which cross-cut abstractions.

One simple cross-cutting concern example is security. Many objects, components and modules within a program

978-1-4673-2560-8/12/$31.00 ©2012 IEEE

may use authentication to check a user's credentials before performing their functional task. When we consider that a system may have hundreds or thousands of structures using security, the problem becomes clearer. Changing the security mechanism or invocation means all these unrelated pieces of code are touched to alter a feature outside their core function.

Aspects are code modules defining three AOP concepts: join points, pointcuts and advice. A join point is a point during the execution of a program, such as a method call or class instantiation. A pointcut is an expression specifying the join point for an aspect. Advice is the code affiliated with a pointcut to execute at the matching join point.

Aspects and AOP are not exclusive to any one particular programming language. The most popular implementation of AOP is AspectJ [3, 4] for Java, though Java has additional options such as Spring AOP [5]. Spring Python [6] is amongst the most popular AOP implementations for Python. In R the r-connect package [7] allows AOP style constructs.

B. The Linda Programming Model

When we talk about coordinating data exchange between processes, we are referring to inter-process communication. There may be shared resources such as a common file or a block of memory space. There may be remote procedure calls; Java Remote Method Invocation and CORBA are two common examples. Message passing between components is another option available. The method used in this particular implementation is the Linda programming model.

Generative communication as the basis for Linda [8]. In generative communication, coordination is achieved through tuplespaces. Rather than communicating directly with one another, programs generate data objects called tuples and add them to an ordered collection known as tuplespace. The core tuplespace operations allow processes to store and retrieve the tuples. The out operation adds a tuple to tuplespace. The read operation retrieves the value of a tuple. The in operation retrieves the value of a tuple and removes it from tuplespace.

Linda originally addressed the need for coordination and communication within parallel programming; its core concepts and tuple spaces are not limited to parallel programming [9, 10]. The functional parts of programs are separable from the coordination and communication between programs; how a scientific model calculates data is independent from how it exchanges data with other models.

III. Overview of Communication Aspects

There are two core parts to our approach of using communication aspects to build software model federations. The first part is managing the data; that means intercepting, filtering and transforming the data. Secondly, there is coordination of the applications, managing the execution and subsequent communication sending the shared data.

A. Data Transformation and Management

Earlier we touched on some general notions regarding data mismatch in a federation. As part of combining software there must be one or more components to bridge the data mismatch discrepancy. Resolving data mismatch can be seen as a cross-cutting concern separate from the core design and computational functionality of the models working together. There must be resolution to the data mismatches for all the objects, components and modules to work together.

B. Process Coordination and Communication

The core concepts of the Linda model and tuple space nicely align with our idea of using AOP to facilitate communication and coordination between programs. We use communication aspects for the scientific federation programs to modularize the communication code, and within these communication aspects they communicate with one another using tuple space. With AOP we uncouple computation from communication, and then within communication itself we uncouple the communication aspects using tuple space.

C. Orthogonal, Uncoupled and Unaltered

Programs communicate and interact with each other through the exchange of data. We want to facilitate this exchange without coupling the exchange mechanism to the functionality of the program. Encapsulating the data exchange uncouples it from the rest of a program, but we also want to do this without changing the original program. Aspect-oriented programming allows us to do both. Rather than alter the code and functionality of the original programs, we allow them to remain the same and use aspect-oriented programming to handle the data handling and sharing.

D. Program Autonomywithin Mu lti-Language Federations

We want to maintain autonomy of the independent programs. Programs can function and execute independent of one another, and the programs do not have to be part of a unified code base. We also want to create federations from programs written in different languages, rather than forcing scientists collaborating together to use the same language or convert programs from one programming language to another. Both these concepts are separation of concerns at a different level; we want to avoid tangling the code of otherwise separable programs, not just avoid internal program code tangling. They are essentially modularity and encapsulation at the program level.

IV. Software Federation Building Blocks

Our example problems process and analyze genomic data from the International HapMap Project [11], plus visualize the results of these computational procedures. We use four common and freely available software applications.

A. UCSC Genome Browser

The UCSC Genome Browser [12, 13] displays genomes with aligned annotation tracks; it can rapidly display any region of interest of a genome, and at any scale. Users enter search terms to specify the position of a region of interest.

B. LocusZoom

LocusZoom [14] is a fast visualization tool for displaying genome-wide association study results. LocusZoom is accessible through a web-interface [15] and as a stand-alone application. Both generate plots of regional association info.

C. FastMap 2.0

FastMap [16] is a fast and efficient method for gene expression Quantitative Trait Locus association mapping in homozygous inbred populations. FastMap 2.0 added gene expression association mapping of heterozygous populations and a graphical user interface [17] for performing association mappings on homozygous and heterozygous populations.

D. SNAP

SNAP [18], or SNP Annotation and Proxy Search, is a web-based tool [19] designed to find proxy SNPs in linkage disequilibrium based on the HapMap Project. In addition to returning annotated proxy SNPs, SNAP can generate regional association plots and graphical plots of proxies.

V. EXAMPLES: COORDINATED AOP COMMUNICATION

A. Software Federation Program Flow

Our bioinformatics federation is composed of four common free programs. Each program in the collaborative tool chain was developed independently and can function independently; each maintains a separate code base. There is a range of application languages, interfaces and platforms.

AOP aspects handle communication between, and execution of, the programs. AspectJ [3] is the AOP language for our coordination aspects. Some subset of those aspects must communicate amongst themselves as well Coordination language handles the role of communication middleware between aspects. We use IBM's TSpaces [20] as the coordination language, a freely available tuple space implementation natively supporting coordination for Java.

B. Example 1: FastMap 2.0 to UCSC Genome Browser

In this first example, the federation link is from FastMap to the Genome Browser. After loading gene expression data and SNP data in FastMap, the user can calculate and plot the association. Upon zooming into an area of the plotted chart, zoomed in data can be sent directly to the Genome Browser. The FastMap code builds the Genome Browser URL, creates a HTTP connection, and opens the URL in the default desktop web browser. The federation link is built-in, coupling the applications. This makes sense here, but in subsequent examples, this is not preferable or feasible.

C. Example 2: FastMap 2.0 to LocusZoom HTTP

For the link from FastMap to LocusZoom, FastMap writes a tab-delimited text file in LocusZoom format. The user can then manually send the file to LocusZoom. In this example federation link, the text file is sent as-is to the LocusZoom web-interface. Instead of manually sending the data file, we want to auto transmit the file. Without access to the FastMap source code, auto transmit functionality cannot be directly added. With communication aspects, we integrate auto sending functionality without altering source code.

We need two things to auto send the data to LocusZoom: the name of the file, and notification when file writing is complete. Below we see code fragments of the first pointcut and advice for the FastMap communication aspect:

```
pointcut getDataFilePath(): (
  withincode(void
    MainGUI.saveChromosomeData(..)) &&
  call(String File.getAbsolutePath())
);

after( ) returning( String str ):
  getDataFilePath() {
    filename = str;
}
```

The function in FastMap to write the LocusZoom format file is `saveChromosomeData`, which is in the Java class `MainGUI`. We know the FastMap application has to get the file path in order to write the file, so we check for calls to the absolute file path within the function and save it in the first advice. Now we look at the second pointcut and its advice:

```
pointcut finishedWritingFile(): (
  withincode(void
    MainGUI.saveChromosomeData(..)) &&
  call(void SwingWorker.execute())
);

after( ) : finishedWritingFile() {
    /* http connect to LocusZoom */
}
```

The second advice executes after the file writing SwingWorker thread completes; the advice sends the file info directly to the LocusZoom web-interface. We build the URL to the LocusZoom web-based form, create a HTTP connection, then post the data and file to the PHP form, without altering the original source code. However, this couples communication and coordination. In our next example, we use a communication aspect to uncouple them.

D. Example 3: FastMap 2.0 to LocusZoom Stand-alone

In this example, the federation link is to the LocusZoom stand-alone application instead of the web-interface. The main change is that rather than building a URL and transmitting the file with HTTP, we directly call LocusZoom with the file and other data on the command line.

Directly calling the LocusZoom application from within the FastMap aspect means we specify execution command line parameters. Suppose we need some processing done, for instance, copying the file to a new location to avoid overwriting by a subsequent FastMap execution. In this scenario, the copying is done on the LocusZoom side.

Introducing an aspect on the LocusZoom side separates FastMap logic from LocusZoom logic. The FastMap aspect must communicate the location of the file to the LocusZoom aspect. We use the coordination paradigm to facilitate uncoupled communication between aspects. In the second FastMap aspect advice, we now send info to tuple space:

```
/* variable declarations */
ts = new TupleSpace(tsName, tsHost);
t1 = new Tuple("lzFilename", filename);
ts.write(t1);
```

Here we see a code snippet from the LocusZoom aspect:

```
/* variable declarations */
ts = new TupleSpace(tsName, tsHost);
template = new Tuple("lzFilename",
                new Field(String.class));
Tuple tuple = ts.take(template);
/* copy file */
```

Whenever FastMap saves data in a LocusZoom format file, the FastMap aspect sends the file info to tuple space. The tuple is read from tuple space via a template in the LocusZoom aspect. The template specifies the tuple name, number of values, and data type of each value. In our case, the tuple has one String value, the file name.

E. Example 4: FastMap 2.0 to SNAP

In this final example, communication aspects handle data computations. Fig. 1 shows the UML for an AOP style [21] diagram illustrating the applications, aspects and tuple space.

LocusZoom data generated by FastMap is also suitable for SNAP. The LocusZoom format is a three-column, tab-delimited file containing SNP marker name, p-value and position. For SNAP, its association plot format is a white-space delimited file containing the SNP marker name and p-value. Its LD plot and text queries take query SNP names as input. A simple filter makes the data ready for SNAP.

```
/* variable declarations */
ts = new TupleSpace(tsName, tsHost);
template = new Tuple("lzFilename",
                new Field(String.class));
Tuple tuple = ts.take(template);
/* filter file data */
/* http connect to SNAP */
```

Here we see a snippet of the SNAP communication aspect. The FastMap communication aspect remains the same, as it only needs to send the LocusZoom file info to the tuple space before proceeding. The aspects are not tied to one another directly; we can swap in and out other application aspects and maintain orthogonal communication.

VI. Related Work

This work is part of ongoing research around software federations, application level coordinated interaction of independently developed heterogeneous applications, plus coordination languages and aspect-oriented programming.

While [22] primarily focuses on using Commercial-Off-The-Shelf software components for creating software federations, it also generalizes around some general tool federation concepts including semantic mismatch, tool invocation and observation, and federation design.

In [23], refactoring using AOP is used to reduce complexity and increase configurability of middleware systems without diminishing runtime performance. Other research investigated large-scale systems and how to reduce internal complexity and external interface complexity [24].

MAML [25] is a language designed to aide scientists, whose expertise is in a field outside of computer science, develop code for computer simulations. AOP separates the model from the observation, visualization and monitoring.

One approach in bioinformatics used aspects to directly exchange data between programs in the federation [26]. AOP aspects intercepted, filtered and transformed the data.

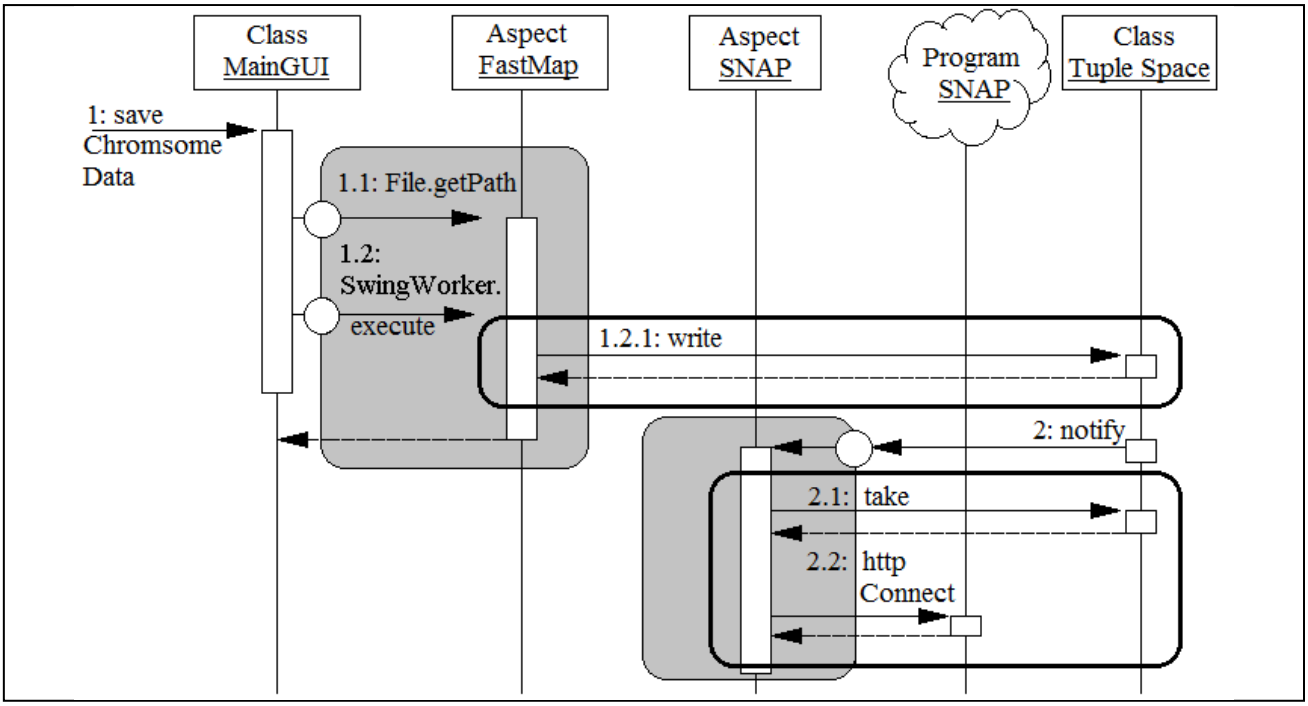

Figure 1. UML diagram of applications and aspects coordinating and communicating through tuple space.

A proposal for parallel scientific code [27] separates the mathematical model from the parallel code execution. Concurrent aspects have also been considered in CEAOP [28], meaning aspects concurrently execute or the aspects and the base programs they advise concurrently execute.

XMLSpaces [29] extends IBM's TSpaces Linda implementation to create middleware for XML applications written using the Microsoft .NET framework. Tuple space systems AspectK [30] and AspectKE* [31] use an AOP system built on tuple space to enforce access control.

When using AOP to internally separate coordination code from computation code, one method delineates data-driven coordination code from event-driven coordination code [32]. Another [33] views coordination as a system of: contracts for interaction, organizers to control the contracts, and roles.

VII. Conclusions

Communication aspects can compose collaborative scientific software model federations. Software federations are an interoperating collection of models exchanging data with one another. Our approach to composing federations is to uncouple communication from computation using aspect-oriented programming and communication aspects. We further uncouple communication between these AOP aspects using coordinated programming, in this case tuple space.

Acknowledgment

Funding for this research comes from the United States Environmental Protection Agency, EPA STAR RD832720, to the University of North Carolina at Chapel Hill, creating the Carolina Environmental Bioinformatics Center.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, "Aspect-Oriented Programming," in 11th ECOOP, M. Aksit, S. Matsuoka, Eds., vol. 1241, 1997, pp. 220-242.

[2] C.V. Lopes, "Aspect-Oriented Programming: A Historical Perspective," in Aspect-Oriented Software Developemtn, R. Filman, T. Elrad, S. Clarke, M. Aksit, Eds., 2004.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, "An Overview of AspectJ," in 15th ECOOP, J.L. Knudsen, Ed., vol. 2072, 2001, pp. 327-353.

[4] AspectJ, http://www.eclipse.org/aspectj/

[5] Spring AOP, http://www.springsource.org/

[6] SpringPython, http://www.springpython.org/

[7] r-connect, http://code.google.com/p/r-connect/

[8] D. Gelernter, "Generative Communication in Linda," in ACM Transactions on Programming Lanaguages and Systems, S. Graham, Ed., vol. 7, issue 1, 1985, pp. 80-112.

[9] N. Carriero, D. Gelernter, "Linda in Context," in Communications of the ACM, vol. 32, issue 4, 1989, pp. 444-458.

[10] S. Ahuja, N. Carriero, D. Gelernter, "Linda and Friends," in Computer, vol. 19, issue 8, 1986, pp. 26-34.

[11] The International HapMap Consortium, "A second generation human haplotype map of over 3.1 million SNPs," in Nature, vol. 449, 2007, pp. 851–861.

[12] W.J. Kent, C.W. Sugnet, T.S. Furey, K.M. Roskin, T.H. Pringle, A.M. Zahler, D. Hausslerm. "The Human Genome Browser at UCSC", in Genome Research, vol. 12, issue 6, 2002, pp. 996-1006.

[13] Genome Browser, http://genome.ucsc.edu/

[14] R.J. Pruim, R.P. Welch, S. Sanna, T.M. Teslovich, P.S. Chines, T.P. Gliedt, M. Boehnke, G.R. Abecasis, C.J. Willer, "LocusZoom: Regional visualization of genome-wide association scan results," in Bioinformatics, vol. 26, issue 18, 2010, pp. 2336-2337.

[15] LocusZoom, version 1.1, http://csg.sph.umich.edu/locuszoom/

[16] D.M. Gatti, A.A. Shabalin, T.-C. Lam, F.A. Wright, I. Rusyn, A.B. Nobel, "FastMap: Fast eQTL mapping in homozygous populations," in Bioinformatics, vol. 25, issue 4, 2009, pp. 482-489.

[17] FastMap 2.0. http://cebc.unc.edu/fastmap86.html

[18] A.D. Johnson, R.E. Handsaker, S. Pulit, M.M. Nizzari, C.J. O'Donnell, P.I.W. de Bakker, "SNAP: a web-based tool for identification and annotation of proxy SNPs using HapMap," in Bioinformatics, vol. 24, issue 24, 2008, pp. 2938-2939.

[19] SNAP, version 2.2, http://www.broadinstitute.org/mpg/snap/

[20] P. Wyckoff, S.W. McLaughry, T.J. Lehman, D.A. Ford, "TSpaces," in IBM Systems Journal, vol. 37, no. 3, 1998, pp. 454.

[21] M. Kandé, J. Kienzle, A. Strohmeier, "From AOP to UML: A Bottom-Up Approach," in Workshop on Aspect-Oriented Modeling with UML, 2002.

[22] J. Estublier, H. Verjus, P.Y. Cunin, "Designing and Building Software Federations," in 27th EUROMICRO 1st Workshop on CBSE, 2001, pp. 121-129.

[23] C. Zhang, H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," in 2nd International AOSD, M. Aksit, Z. Choukair, Eds., 2003, pp. 130-139.

[24] A. Colyer, A. Clement, "Large-scale AOSD for Middleware," in 3rd International AOSD, G. Murphy, K. Lieberherr, Eds., 2004, pp. 56-65.

[25] L. Gulyás, T. Kozsik, "The Use of Aspect-Oriented Programming in Scientific Simulations," in 6th Fenno-Ugric Symposium on Software Technology, J. Penjam, Eds., 1999, pp. 17-28.

[26] D. Stotts, K.L. Lee, I. Rusyn, "Supporting Computational Systems Science: Genomic Analysis Tool Federations Using Aspects and AOP," in 4th ISBRA, I. Mandoiu, R. Sunderraman, A. Zelikovsky, Eds., vol. 4983, 2008, pp. 457-469.

[27] B. Harbulot, J. Gurd, "Using AspectJ to Separate Concerns in Parallel Scientific Java Code," in 3rd International AOSD , G. Murphy, K. Lieberherr, Eds., 2004, pp. 122-131.

[28] R. Douence, D. Le Botlan, J. Noyé, M. Südholt, "Concurrent Aspects," in 5th International GPCE, 2006, pp. 79-88.

[29] R. Tolksdorf, F. Liebsch, D.M. Nguyen, "XMLSpaces.NET: An Extensible Tuplespace as XML Middleware," in 2nd International Workshop on .NET Technologies, V. Skala, P. Nienaltowski, Eds., 2004.

[30] C. Hankin, F. Nielson, H.R. Nielson, F. Yang, "Advice for Coordination," in 10th International Conference on Coordination Models and Languages, D. Lea, G. Zavattaro, Eds., 2008, pp. 153-178.

[31] F. Yang, T. Aotani, H. Masuhara, F. Nielson, H.R. Nielson, "Combining Static Analysis and Runtime Checking in Security Aspects for Distributed Tuple Spaces,". in 13th International Conference on Coordination Models and Languages, W. De Meuter, G.-C. Roman, Eds., 2011, pp. 202-218.

[32] S. Capizzi, R. Solmi, G. Zavattaro, "From Endogenous to Exogenous Coordination Using Aspect-Oriented Programming," in 6th International Conference on Coordination Models and Languages, R. De Nicola, G.L. Ferrari, G. Meredith, Eds., 2004, pp. 105-118.

[33] A. Colman, J. Han, "Coordination Systems in Role-Based Adaptive Software," in 7th International Conference on Coordination Models and Languages, J.-M. Jacquet, G.P. Picco, Eds., vol. 3454, 2007, pp. 63-78.