RECONFIGURATION OF HIERARCHICAL TUPLE-SPACES: EXPERIMENTS WITH LINDA-POLYLITH

Gilberto Matos James Purtilo

Computer Science Department and Institute
for Advanced Computer Studies
University of Maryland
College Park, MD 20742

ABSTRACT: A hierarchical tuple-space model is proposed for dealing with issues of complexity faced by programmers who build and manage programs in distributed networks. We present our research on a Linda-style approach to both configuration and reconfiguration. After presenting the model used in our work, we describe an experimental implementation of a programming system based upon the model.

This research is supported by the National Science Foundation under contract NSF CCR-9021222.

1 INTRODUCTION

While the specification of the algorithm is usually accepted as a good representation of a sequential program, complex distributed systems are rarely represented using one elaborated model. The complexity of representation of distributed systems lies in the existence of multiple processes, which can be arbitrarily synchronized or completely asynchronous, and whose communication patterns are allowed to be dynamically reconfigured during the execution of the program. The main requirement for a model which can be considered to be simple and easy to use is the static structure. The purpose of this paper is to describe one static structure which captures the complexity of distributed programming systems. This model of distributed programming systems can provide information which can be very useful in the maintenance of software, thus increasing reusability and productivity in software engineering. This model is based on a simple static structure, but it is designed to be able to model any type of distributed applications.

In Section 2, after a brief background overview on the tuple space concept, we will define the requirements for a static hierarchical representation of a distributed system. This set of requirements is geared towards allowing static analysis of the communication patterns in the system, and it imposes certain constraints on the design of a program, but indirect ways for bypassing these constraints will be discussed in 2.3.

The Section 3 contains a discussion of different software manipulation techniques which can be supported by an automatic compile-time analysis of the program, based on the proposed model for the system. For this analysis we will isolate several basic types of communication patterns and analyze the methods for the basic software structure manipulation, such as addition of modules, replacement of modules the merging of complete applications. We will also show how different types of sequential programs can be modeled using this representation, and what are the benefits of such representation.

Section 4 illustrates some practical considerations of the proposed model, based on a Distributed File Server implementation. Performance of the system is also discussed with a look at the scalability issues for our current implementation and possible improvements.

2 TUPLE SPACE TREE

The purpose of this section is to briefly review the background of Linda style computing, and then to introduce our model of hierarchical tuple spaces.

2.1 LINDA AND THE TUPLE SPACE CONCEPT

Linda is a Distributed Programming Language which is based on the associative communication concept, and total dynamic reconfigurability. The communication in Linda consists of tuples, which are messages of data without any address information. When a tuple is matched by a request for receiving a tuple, then the communication is effectively performed.

Syntax of Linda can be found in detail in [CG89a]. In short, Linda consists of four primitives and some derived commands. Three main commands handle the communication and one command covers the creation of processes. The commands are the following:

1. out - (send message)
2. in - (receive message)
3. rd - (receive message and leave a copy in tuple space)
4. eval - (create new process(es))

The out command creates a tuple which joins the tuple space and waits for some process to accept it. The receiving is performed by matching the tuple in the tuple space with the format and specified values in the receivers request. Message can be deleted from the tuple space or left in it, depending on the receiving operation. The eval command creates a template which is an active tuple, a set of requests for computations that should be performed in order to compute the values in the tuple. When all values of the active tuple have been computed, it becomes a normal passive tuple, no different from other messages in the system. Figure 1 shows a simple solution of the dining philosophers problem using Linda.

```
void main()
{ int i,num=5;
  out("NUMBER",num);
  for(i=0;i<num;i++){
       out("FORK",i);
       eval(philo(i));
       if(i>0) out("TICK");
  }
}
```

```
int philo(int i)
{ int num;
  rd("NUMBER",?num);
  for(;;){
    work();
    in("TICK");
    in("FORK",i);
    in("FORK",(i+1)\%num);
    eat();
    out("TICK");
    out("FORK",i);
    out("FORK",(i+1)\%num);
  }
}
```

Figure 1: Linda Example – This is a Linda solution for the dining philosophers problem modified from [CG89a].

Linda gives complete dynamic reconfiguration capabilities. Every process executes some function and "dies" when the function is computed, and any process can initialize the creation of a new process. The message addressing is emulated by matching the formats and values of the message attributes, whose values can be changed dynamically at runtime, thus changing the communication pattern. With two derived functions (inp and rdp) which handle conditional receiving of messages, Linda is capable of performing any distributed computation.

2.2 HIERARCHY OF TUPLE SPACES

The concept of tuple space, as used in Linda, represents a global associative shared memory used for communication. Instead of explicit addressing, associative communication uses the format of the message and its content to match arriving messages with requests. When the matching is successful, the message is forwarded to the process which sent the request. This type of communication has implicit addressing based on the content of the message which makes complex communication patterns easy to implement.

Tuple space is a very simple, yet powerful, concept for modeling distributed programs, but it doesn't support the creation of large systems because all the processes in the system have access to one tuple space containing all the messages (see example 2). Dividing the global tuple space into disjunct subspaces allows the programmer to direct the communication of processes in the system to particular subspaces. Any subspace which is reachable only from a certain subset of processes in the system can be considered to be their local tuple space, while the rest of the system can reach it only indirectly through communication with some of the local processes. Isolating a part of the system in this way is used extensively in object oriented programming and abstract data types.

Access to more than one tuple subspace is necessary for some of the processes, because otherwise the system would be divided into isolated subsystems. Message addressing requires the tuple subspaces to be organized according to their accessibility to the processes in the system. Hierarchical organization of tuple subspaces from local to global level allows the introduction of a simple static structure for modeling the system; a simple structure for this hierarchical model is a tree which has tuple subspaces as nonleaf nodes and processes as leaves. The root of the tuple space tree is the global tuple subspace, which is accessible to all the processes in the systems; all the other nodes are local tuple subspaces which allow access only to processes which are their descendants.

Tuple space tree has a very simple addressing scheme which is used for directing messages to tuple subspaces for matching; In addition to the tuple attributes, the process needs to specify the number of levels that the message should climb towards the global tuple subspace. Static addressing of the message allows compile-time checking to find all the processes that can potentially communicate through a given tuple subspace. Allowing the compile-time checking of communication between processes is sufficient reason for making static message addressing a requirement in this model. Methods for removing the need for run-time tuple space addressing will be addressed in 2.3.

Creation of new processes in Linda is done transparently to communication, by making an active message which requires the creation of new processes which will compute the values in the tuple.

The tuple space tree requires specifying where are the processes going to be executed and where to send the message after it is computed; Restricting the creation of processes to the local tuple subspace of the calling process removes the need to specify for each process in the active tuple where it should be executed, and allows static checking of processes which can be created within a tuple subspace. Methods for reducing non-local process creation to local will be discussed later.

```
void main()
{ int i,num=5;
  out("NUMBER",num);
  for(i=0;i<num;i++){
    out("FORK1",i);
    out("FORK2",i);
    eval(philo(i,"TICK1","FORK1"));
    eval(philo(i,"TICK2","FORK2"));
    if(i>0) out("TICK1");
    if(i>0) out("TICK2");
  }
}
```

```
int philo(int i, char *tick, *fork)
{ int num;
  rd("NUMBER",?num)
  for(;;){
    work();
    in(tick);
    in(fork,i);
    in(fork,(i+1)\%num);
    eat();
    out(tick);
    out(fork,i);
    out(fork,(i+1)\%num);
  }
}
```

The dining philosophers problem has a very simple solution in Linda, but making it slightly more complex illustrates the problems with duplication of messages. Suppose we want to add another set of philosophers and another table for them. The synchronization of both groups should be done independently and no messages should be shared by both groups. This means that the groups use different sets of messages which can't be matched. One way to use different messages is to replicate the code for the *philo* function with different values for messages, but this duplicates the source, and makes later changes in the code harder. A better solution is passing the descriptors of messages as formal parameters, but this method makes the communication hard to analyze. Problem lies in the fact that we are putting structural information about the application inside the messages in a desorganized way, and this implied structure can't be extracted from the source code at compile time.

Figure 2: Linda solution for 2 groups of philosophers

2.3 RESTRICTIONS OF THIS MODEL

This model explicitly divides the tuple space into disjunt subspaces, where the set of processes that can access any tuple subspace can be found at compile time. This information is specially useful in work with complex systems, when it is hard to remember all the types of messages in the system or any of his parts. The restrictions that are introduced in the description of the model are required in order to make the model simple and static, even though it represents a dynamically reconfigurable system. This model of distributed programing systems also has in mind higher performance of communication due to the smaller size of the tuple spaces.

Restricting the process creation to the local tuple subspace is important because it enables the static analysis to find all the potential processes in any given tuple subspace, and process creation remains structure independent. This requirement doesn't really disable any inter-subspace process creation, but only requires that the creation be requested by a message to a local process. When many requirements for inter-subspace process creation exist in the system, the creation-request messages will introduce too many communication dependences in the system. Such parts of the system should be considered for maintaining in one local tuple space, where the process creation would be handled directly; additional analysis should be performed to find out if restructuring can improve the situation.

The requirement for static message addressing is necessary for enabling the compile-time analysis of communication in the system. The constraints which are introduced by this requirement are not critical, and they shouldn't even show up except when applications written using some different model are being implemented according to this one. If some communication commands in the system had a variable address field, they could potentially establish communication through any level of tuple spaces and compile-time checking would be unable to determine the processes which will use any particular message format in a given tuple space. This would require a conservatve approach in the estimation of the set of processes and the resulting set would have to be bigger than in the case of static addressing. Another problem with the dynamic message addressing is that the program would have to be written having in mind a particular tuple space tree, which would severely limit the possibilities of reconfiguration of the tree.

Substitution of dynamic by static message addressing can be done in two simple ways. One way would be to merge the involved messages into one tuple space, and then address the messages statically. This may require adding some synchronization mechanizms for different processes or adding selection values to messages in order not to introduce unwanted message matches. Second way is to separate a dynamically addressed communication command into a series of static commands which are activated according to the value of the addressing variable. In case of a reconfiguration of the tuple space tree, the static addresses can be changed at the source level to satisfy the new tree structure.

This enables the compiler to make a precise estimation of the processes which will send or receive messages of a certain format to any tuple space, and which could therefore communicate during the execution.

3 SOFTWARE MANIPULATIONS

The model just introduced has a number of important simplifications for the application structure which allow for significant increase in clarity of programming, and easier analysis. Both of these aspects are important for the maintenance and continued development of a complex software system.

The following analysis will demonstrate how some simple methods can help the programmer to detect the parts of the application which he needs to change or just check for correctness, after some previous change. As in the previous chapters modules will refer to sets of processes which have a local tuple space, although it can consist of one process which doesn't necessarily use that local tuple space.

The analysis is based on the afore mentioned constraints on the reconfiguration, and on the statically addressed associative communication. There are several types of dataflows in the application, which are interesting to us because of the different methods for adding or deleting some processes which access them. The authors of [CG89b] claim that parallel programs can be done using a small number of process types which communicate through the tuple space. Our idea is that the number of communication patterns should also be limited. Having a small number of communication patterns enables us to make rules for changes on every pattern. The following list should be sufficient to capture all complexities of distributed programs. Since a sequential program can in theory do everything that a distributed one can, then a distributed program with limited types of communication is also able to do everything as ones with unlimited communication.

1. Asynchronous communication of multiple processes.
   Any number of processes is allowed to read or write messages at any time, and the order of messages is not important.

2. Synchronized one-one communication
   This type of communication exists between two processes, when each process sends a message, and then waits for the response from the other process. Nonrecursive procedure calls can be modeled using this type of communication.

3. FIFO Communication with multiple processes.
   In general this type of communication requires synchronization on the writing and reading end because messages need to have some ordered identification, and processes need to read and increment the identification before performing the read or write operation. The number of processes which are accessing this message flow is allowed to vary during the computation.

4. Other types of communication which usually encompass some implicit synchronization. These patterns should be analyzed more carefully to see how the synchronization is performed, and how the desired change in the structure of the application would interfere with the existing synchronization pattern. Reduction to some of the previous patterns should be possible, which will make this analysis complete.

Some of these communication patterns already have some kind of explicit process synchronization, while for others it is implied or nonexistent and unnecessary. These differences in the communication patterns will result in differences in the software manipulation operations , depending on the types of communication that they have to interfere with.

- REPLACEMENT OF A MODULE
  When something is to be changed inside one module, it can be done without any regard to the system if the change concerns only data which will not interfere with communication or synchronization of any other process. In such cases we are dealing with changes in the data processing within the module, and no external changing is required. The checking is

necessary on all of the modules which can get the data from this module, and use it to determine the control flow, if it can affect communication.

If some of the communication interfaces is to be changed, then the checking has to include the synchronization of the specific dataflow, and the message values. For the case of asynchronous communication and FIFO synchronized communication, change in this module can only raise problems if it is the only module on the read or write side. On the other hand if the module has a 1-1 synchronized communication with some other module, then a change in this interface requires a change in the respective interface in the other module.

- ADDITION OF A MODULE
  Addition of a new module in the proposed representation is pretty straightforward based on the static tuple space addressing. When the module is specified and the internal and external data dependencies are specified, it can simply be plugged into all the dataflows, and then a simple checking routine can determine if some synchronization needs to be added. Adding a new process any of the multiprocess communication patterns doesn't require creation of new synchronization, but it may require the module to be adapted to the synchronization requirements of the interface. Adding a new process to the 1-1 synchronized communication will probably require some changes in the other two modules, in order for the communication to work correctly.

  The above analysis requires that the new module has access to all of the communication links it needs, but it doesn't have to be true in the structure of the application. Some of the communication interfaces can be in two different subtrees of the structure, and no process from any of the local levels can access both of these interfaces. This requires moving at least one of these interfaces, to some tuple space which is common ancestor for the current tuple spaces, which doesn't have any conflicting interface already. Only after this the new module can be linked to both of these interfaces. The movement of the interface will inevitably require a change in all modules that access it, and it consists of a static change of addressing for the given interface, and possibly for other interfaces which are used for synchronization over the first interface.

- MERGING OF TWO APPLICATIONS
  The simplest conceptual way of merging two applications is to add a new level of global tuple space, and then to link them through it. This kind of linking requires some of the interfaces in the applications to be changed and addressed to the new tuple space, or even more probably it will require the addition of new modules which will do the communication through the global level, and at the same time communicate at the non-global levels with the existing modules, thus providing themselves as local interfaces which can be transparent to the rest of the application.

  Another way of merging two applications is to add one application as a module to the other. This requires that the added application gets adapted to the interfaces to which it is to be connected. This merge operation can be done in a cleaner way if instead of the whole application , one module is added with the correct interfaces , and then the module is replaced by the whole application. Again some local processes can be added to the application to provide the global interface instead of changing existing modules.

3.1 REPRESENTATION OF SEQUENTIAL PROGRAMS

Sequential programming languages can also be modeled according to the proposed structure, and their representation according to this model can facilitate their evolution towards distributed systems. The transformation of sequential into distributed programs can be divided in three main phases: structure generation, structure linking and synchronization. In this transformation procedures will be transformed into processes, and variables into messages. All the processes will be synchronized for only one active process at any time.

1. Structure Generation
   The creation of the tuple space structure is performed according to the hierarchy of variables and procedures in the original program. This process can be tied to any or both of the hierarchies in the program.
2. Structure Linking
   After the creation of the structure, functions and variables from the original program are linked to the nodes of the structure, according to the rules used for creating the structure.
3. Synchronization
   In order to make the procedures into processes, they have to be initialized independently, and then wait for a procedure request to arrive in form of a message. After their operation is finished, procedures send a message with the results, and unblock the caller function which was waiting for that message. If the procedure calls can be recursive, then the messages need to carry the identification of the call, and creation of additional processes has to be implemented.

4 EXPERIMENTAL RESULTS

A series of experiments has been implemented using the proposed model of distributed applications. These experiments have provided us with valuable insights into the effectivness of this method in implementation of complex distributed systems. The experiments have also shown that explicit partitioning of the tuple space is a powerful method for optimizing the execution time of distributed programs with associative communication. One implementation of a distributed file server will be discussed in order to provide the reader with a practical view of this methodology on a slightly more complex problem than the dining philosophers which have been used so far.

4.1 DISTRIBUTED FILE SERVER

An implementation of a distributed file server is a good example to illustrate some characteristics of the proposed model and to perform some performance tests. This read only file server consists of one process working as catalog and several servers which can be placed on physical file servers. The users view the file server as one entity to which they can send their requests without knowing anything about the file mapping. All the file requests are received by the catalog, and he decides to which file server to send them. The local file servers create a new process for every file that has to be transmitted, which enables parallel serving of any number of requests. Figure 5 gives a dataflow diagram of this system.

The tuple space is used for both communication and information storage in this system. The "files" are stored as sets of tuples, and local file servers read them and send them to the users. The tuple space is partitioned into three subspaces, two of which are local for the file servers and are used for storage, and one which is global and used for communication between users and the file server system. The data transfer protocol which is used in this example consists of one message specifying the length of the file, which is followed by the necessary number of messages containing one record of the file. For simplicity reasons, the user processes are created in the local subspaces for servers, but they only communicate with the server through the global tuple space. Because of the static message addressing requirement, it is possible to determine that user processes can't interfere with the state of the local tuple spaces even if they are using messages of same format for communication, as is the case in this example.

4.2 IMPLEMENTATION TOOLS

The implementation of the described system was done using two basic tools, which were chosen because their complementary strengths could be appropriately combined to enhance the overall performance of the package. The associative communication and the dynamic reconfiguration were provided by a local Linda implementation. The Polylith software bus was used to provide static communication between the Linda modules, and to allow multiprocessor and multiarchitecture work.

Our Linda system is a single processor, multiprocess implementation of all the Linda defined functions on a subset of message formats. The constraint to single processor has been chosen because it allows the communication to be performed by high speed channels, such as message queues. The constraint to a subset of formats was driven by the desire to lower the cost of message matching while maintaining as much as possible the linda expressivness. Both of these constraints are not serious in the sense that they preserve all the dynamic reconfiguration and associative communication requirements of Linda. The matching of messages in this system is performed by one server process, which communicates directly with all the clients.

The Polylith system [Purt] contains a set of procedures which allow statically configured communication between processes running on one or more processors which are not necessarily of the same architecture. The statically addressed communication in Polylith doesn't put high requirements to the communication protocol, and it is therefore simple and very efficient. The Polylith system is used in this system as a base for transparent implementation of the Linda system on multiple processors, thus resulting in a very powerful multiprocessor implementation of Linda.

The structure of a distributed system defined by the hierarchical structure of the tuple space, is implemented using Linda and Polylith as illustrated in Figure 6. Each tuple subspace is served by one Linda system, while Polylith provides the communication between the Linda servers at adjacent levels. Without loss of generality, the Linda subsystems are divided into three types, leaves, intermediate subspaces, and global subspace. The only difference between them is in the types of underlying communication they use because the leaf level is the only level with local reconfigurable processes, while intermediate and global tuple servers only accept messages through polylith from other tuple servers. Addition of client processes to the intermediate levels can be accomplished by adding local leaf tuple servers as their clients.

The communication in this system is performed by indirect connections through the tree, and not by direct addressing to the destination server or process. This puts an additional load on the performance of the system, but it still shows improvements over simple linda implementation. Since one of the requirements of this model is static addressing of messages to tuple subspaces, it is possible and desirable to implement this system using direct connections which would result in additional performance improvements.

4.3 PERFORMANCE CONSIDERATIONS

One of the biggest reasons for low eficiency of associative communication systems is the message matching process which has to compare the format and some attribute values for all the messages and requests in the tuple space. The cost of this operation tends to be linear in the number of messages with matching format. The static message addressing and tuple space partition result in less potential matches of messages and requests, thus lowering the overall processing cost for the application. The explicit division of the tuple space which is done in the file server example reduces the size of the tuple spaces, which lowers the processing time for the tuple matching.

The file server was tested for time in three different implementations, two of which were with partitioned tuple space while the third was the control example executing on a single tuple space. All three implementations were identical in terms of the Linda code being executed and had the same communication load to the tuple space. The partitioned tuple space implementations differ only in the number of processors, one executes on a single processor, while the other is distributed over 4 processors. In the distributed example, the three tuple servers and respective processes were on the same processor, and another processor was used for the Polylith bus control process.

On Sun Microsystems Sparcstations, total CPU time for the single tuple space application was determined to be 35 seconds, while the three tuple server processes in the partitioned application took 30 seconds CPU time executing on a single CPU. This shows that the partitioning of tuple space in this case gives almost 20% improvement in performance of the communication system. This speedup doesn't take into account the overhead which is spent on sending messages through Polylith, and the Operating System overhead for handling all the processes in the system. The CPU time which was used for the communication between these tuple servers can be estimated based on the time that the Polylith bus server was active. The bus server used 14 seconds of CPU time, and most of that time was due to communication because routing in the system is trivial with just 3 modules. From the available data, we estimated that half of that CPU time was required for the tuple servers to send their messages, which would reduce the message matching

CPU time to less than 50% of the single tuple space implementation. The wall clock time for the execution of both examples was approximately the same with small oscillations, around 90 seconds.

The multiprocessor implementation was identical to the single processor system, and the difference in communication cost was very small because the execution took place on several local processors. The wall clock time for this implementation was 30 seconds, which means that this partition of the tuple space gave very good load balancing on the tuple spaces. The sum of CPU times for the tuple servers in this case is 21 seconds which is an almost 50% speedup over the single tuple space implementation. This time still includes some communication overhead, which is likely to lower the effective tuple space processing time to under a half of the original time.

4.4 SCALABILITY OF THE APPLICATION

As mentioned before, this model of distributed applications is very appropriate for changes in the application, with special accent on replication of parts of the application. As an illustration of this aspect of the tuple space partitioning we will replicate the Distributed File Server within one application, and show what kind of timing results were obtained.

The structure of this application is very simple and derived from the original sructure by adding a new global level of tuple space which will not be used, and whose only utility is to be a dummy root for a tree. Each of the File Servers is replicated as a separate subtree, and their execution is still identical to the original example.

The only change in the source for this application has to do with the tuple space tree structure specification and initialization functions. There the two subtrees have to be made descendants of the dummy global tuple space, and the application is ready to execute.

This application has been distributed over 7 processors,so that every tuple server with his client processes was given a single processor, and the Polylith Bus control process was alone on one processor. Since all the other processes were replicated and distributed over the network, only the Polylith control process remained as the bottleneck for this communication intensive application. Nevertheless, this bottleneck contributed with only a 30% increase in the wall time for the execution.

5 CONCLUSION

In this paper we first gave a set of experimental structured distributed systems, which illustrate the potential simple methods of doing some maintenance or development work on the structured distributed applications which obey to certain constraints. A proposed structure with a set of constraints is then discussed in more detail, with methods for performing some basic operations. This set of operations is based on a relatively small set of communication patterns, which we consider to be able of modeling any other pattern. Furthermore, we hope to define a method which will be able to create applications using only these communication patterns, which would give the basis for further development based solely on these communication patterns.

It is our belief that these methods can be incorporated into a software development system, which would then be able to help the programmers on two distinct levels. Automatic placement of interfaces in different tuple spaces , together with the maintenance of coherence with the static addressing at the source level, could be performed as a background activity without involving the programmer. The interactive part would be in charge of detecting potential communication or synchronization dependencies in the structure, and the programmer would be warned of them (grammar?) in order to check them and change them if necessary.

This model of Distributed Programming Systems can also contribute to increased performance, because it detects the communications which can be executed concurrently, and therefore parallelism of the system is improved. Another way of increasing the performance is by the detection of processes which should be physically close together in order to use the communication in the most optimal way.

The generality of this approach enables it to model any type of software, and furthermore it can be naturally enriched by adding the necessary rules for some specific systems (functional languages, sequential languages). We hope that the use of this method can have great impact on the software development and also on reusability because it makes it much easier to analyze any programming system, which is necessary for reusing if it doesn't have a strict specification.

BIBLIOGRAPHY

[CG89a] Carriero, N., Gelerntner, D. Linda in Context. **Communications of the ACM**, (April 1989), vol. 32, no. 4.

[CG89b] Carriero, N., Gelerntner, D. How to Write Parallel Programs: A Guide to the Perplexed. **ACM Computing Surveys**, (September 1989), vol. 21, no. 3.

[Purt] Purtilo, J., The Polylith Software Bus, To appear, **ACM TOPLAS**. Currently available as *University of Maryland CSD Technical Report 2469*, 1990.

```
tuplespace global
 {
   tuplespace local1
       {
         init_process main();
         init_tuple("NUMBER",5);
       }
   tuplespace local2
       {
         init_process main();
         init_tuple("NUMBER",7);
       }
 };
```

```
void main()
{ int i,num;
  rd(0,"NUMBER",?num);
  for(i=0;i<num;i++){
    out(0,"fork",i);
    eval(0,philo(i));
    if(i>0) out(0,"tick");
  }
}
```

Division of the tuple space into subspaces allows the replication of philosopher "parties" provided that each one is communicating through a different subspace. Implementation requires the explicit specification of the structure of tuple spaces (fig) which will allow the analysis of communication in the system. The source code of the application can remain the same as in (fig) with the added address information for all the communication commands. In this case all messages go to the local tuple spaces so the only addition is the *0* as the first parameter. The init_process and init_tuple commands specify the state of the tuple space and the processes which should initialize the system. This alows us to introduce position specific information without different versions of source code.

Figure 3: Solution with partitioned tuple space

```
void work()
{
 in(1,"TICK");
 talk();
 out(1,"TICK");
}
```

```
tuplespace global
  {
    init_tuple("TICK");
    tuplespace local1;
    tuplespace local2;
  };
```

Figure 4: additional improvement with partitioned tuple space

In example 1, we created two groups of philosophers where each group has its own table where they need to get syncronized. When they all work together, only one philosopher can talk at any particular time. Using the tuple space tree concept, these philosophers can be synchronized through a separate tuple space which is ancestor to both local tuple spaces. Synchronization of all philosophers can be done by taking a single tuple from the tuple space, and the name of this tuple can be "TICK", the same that is used for the local tuples. Static addressing makes these messages different and they can't possibly be interchanged.

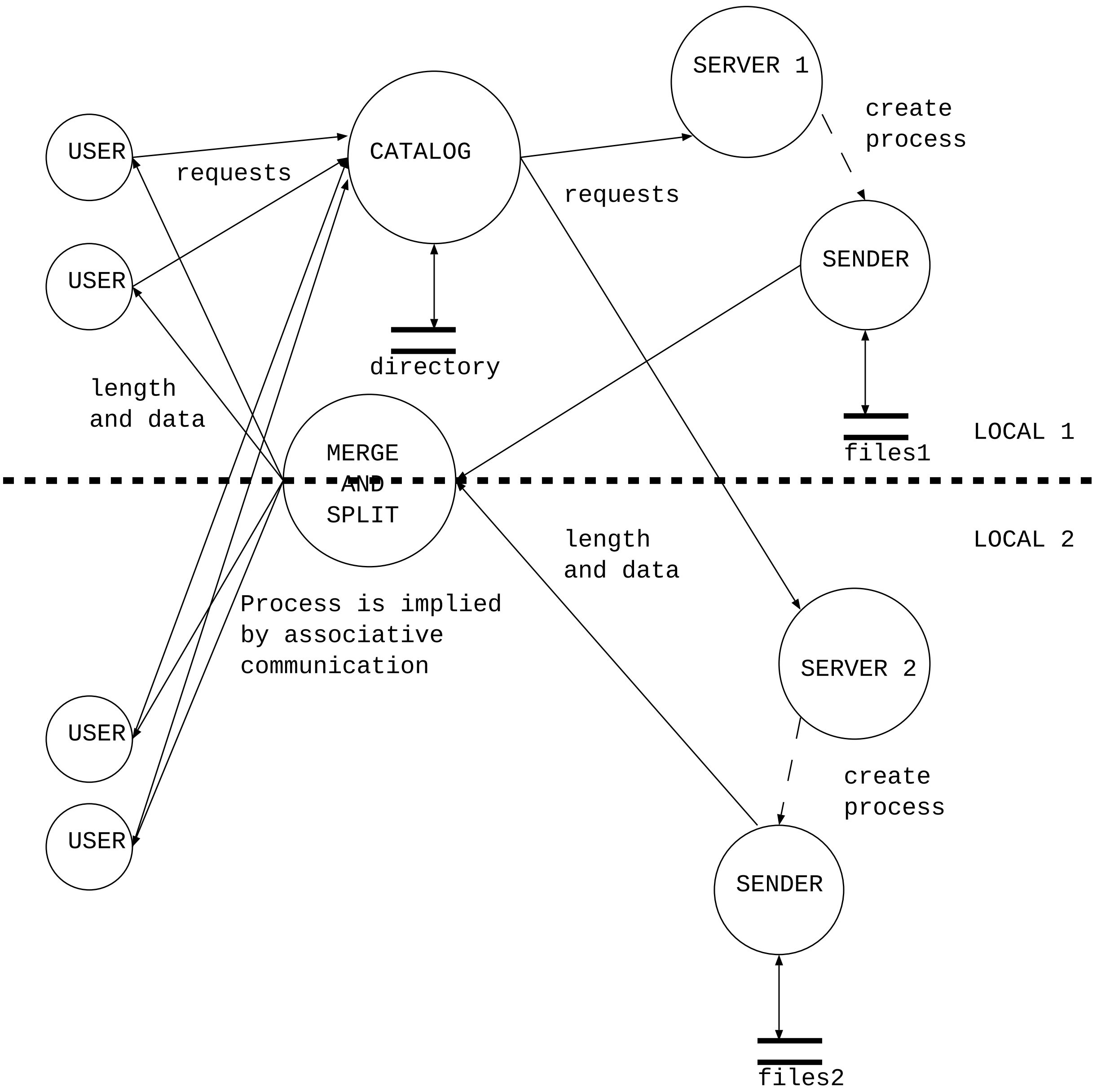

Figure 5: Distributed File Server

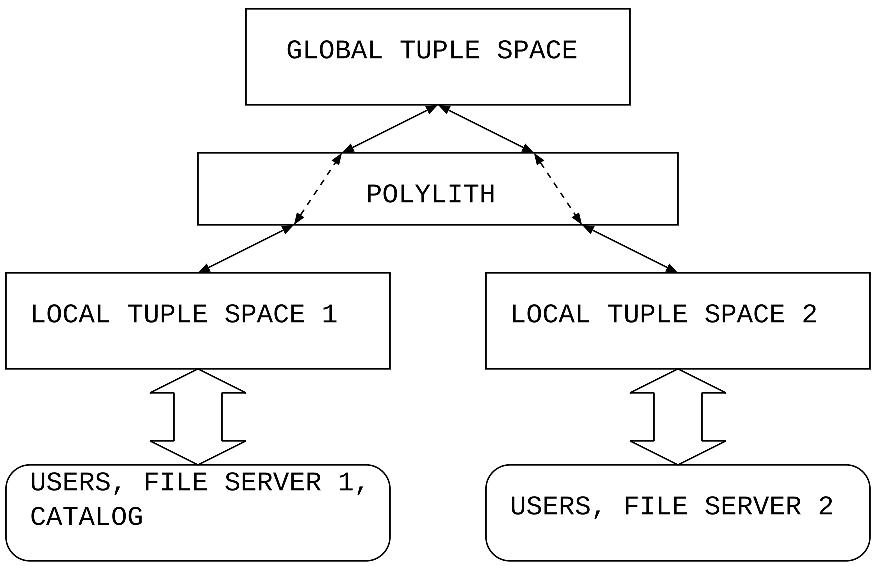

Figure 6: Structure of the Application