Designing an Open-ended Distributed File System in Aster

V. Issarny, C. Bidan, and T. Saridakis
IRISA
Campus de Beaulieu, Rennes Cedex 35042, FRANCE

Abstract

In today's field of distributed computing there is a need for platforms allowing the easy development and automated configuration of applications consisting of heterogeneous software modules and running on heterogeneous hardware platforms. ASTER is a distributed programming system that permits the programmer to construct a distributed application regardless to the physical characteristics of the hardware and software components that will be used. It also customizes the distributed runtime system automatically in order to satisfy application needs. We demonstrate the capabilities of ASTER by proposing a design approach for a platform integrating existing file systems into a *federated distributed file system* ($\mathcal{FDFS}$). Our system allows the specification of application requirements concerning *Quality of Service* (QoS), and based on them it builds the interconnection environment automatically.

Keywords: *Distributed programming, Dynamic reconfiguration, File systems.*

1 Introduction

A number of solutions have been proposed to provide the programmer with an environment for building distributed applications from heterogeneous software components and having them running over a set of heterogeneous hardware components. This area of research deals with the dynamic evolution of the distributed application structure and with compromising the demands for Quality of Service (QoS) and the various servicing policies required by application components. Our approach to the distributed programming environment challenge is called ASTER [1, 2]. It is a software interconnection system that follows the model of *configuration-based* distributed programming. This model requires that the interconnection of the application modules is decoupled from their implementations. Our work is based on the study and experience gained from similar distributed programming environments like POLYLITH [3], CONIC [4], and DARWIN [5].

ASTER consists of *i)* the ASTER language which allows the programmer to describe the interfaces of its modules, and the interactions among them, and *ii)* the run-time system or *software bus* which is customized to satisfy the needs of a given application (e.g. security on exchanged data, availability of application modules, characteristics of the communication channels, etc.). The ASTER language combines the functionality of the *Interface Definition Language* (IDL) of OMG [6] with the flexibility of a *Module Interconnection Language* (MIL) [3, 4] in declaring module interfaces and specifying the interconnections among them. Moreover, it provides the ability to express the properties required/provided by application modules. An ASTER software bus consists of a *base-bus* providing basic communication mechanisms (e.g. send and receive) and a number of *system components* responsible for customizing the interconnection environment to meet application needs. A fundamental goal of our work is to determine a formal way for specifying the functionality of a system component and the way it interacts with its environment so as to allow automatic configuration of the software bus according to the required properties.

In this document, we demonstrate the capabilities of ASTER by an example: the composition of a *Federated Distributed File System* ($\mathcal{FDFS}$) from existing file systems. In this example, we address the issues of *dynamic binding* and *dynamic reconfiguration* of the application's interconnection structure. We propose a solution for gradually transforming a set of independent file systems, to a fully functional distributed file system which federates the service policies of its heterogeneous base elements. $\mathcal{FDFS}$ will be able to deal with diverging client requirements for QoS including issues of security, availability, timeliness, accounting, monitoring, billing, etc.

This paper is structured as follows: section 2 gives an overview of the $\mathcal{FDFS}$ in the framework of ASTER. Section 3 focuses on the problem of re-structuring the interconnection system during run-time, in order to satisfy dynamic binding requirements. We present related work in section 4, and we summarize our contribution and refer to future work in section 5.

2 What is the $\mathcal{FDFS}$

Several attempts have been made for constructing a distributed file system. Some of them take advantage of specific hardware (e.g. [7]), while others base their approach on special OS. (operating system) characteristics (e.g. [8]). Moreover, there are cases where newly proposed distributed file systems are built over existing ones (e.g. [9]) by introducing a higher layer for a number of file-specific manipulations (e.g. declustering, data format translation, etc.), or by adding levels of abstraction for file system operations (e.g. data/name caching, file position within the system, etc.). In each case, design decisions are supported by strong arguments (e.g. performance, availability, security, cache-coherency, etc.). The common point in all approaches is the designers' desire to achieve sharing facilities by integrating a number of physically separated elements into a single distributed file system which can provide transparent file access.

Our standpoint. To distinguish our approach from those mentioned above we state that we do not propose a new file system architecture in terms of a new OS. or in terms of a new mechanism satisfying specific requirements related to file distribution, access, security, availability, etc. We suggest the construction of an open-ended distributed platform which can accommodate any number or type of existing file systems and provide the remote client with no less properties than those offered by the base constituents. $\mathcal{FDFS}$ consists of a number of clients, a number of Local File Servers (LFSs) which eventually serve clients' requests and a distributed platform responsible for interconnecting them. Each LFS is characterized by the services that it provides (e.g. availability, performance, security, etc.). The interconnection environment also provides a number of services (e.g. naming policy, secure communication channels, etc.). The structure of the interconnection environment can potentially grow in a hierarchical manner, and new services can be built on top of existing ones. In this paper we discuss the design of the interconnection platform for the $\mathcal{FDFS}$ that realizes the *global naming* service which is essential for transparent file access in such an heterogeneous ensemble.

Transparent file access. The naming issue is a fundamental one in distributed systems. In a distributed file system, the naming policy determines the degree of file access transparency. Different approaches for naming have been used in this area, varying from attaching a remote name space to the local name space (e.g.[10]), to integrating all local name spaces into a single global one (e.g. [9]). $\mathcal{FDFS}$ adopts the latter approach, i.e. it assumes a single, global root for the file system tree without any explicit assumptions on the mechanism used to achieve it (i.e. static/dynamic name tables, name caching, etc.). In the remainder we consider direct file operations (e.g. *open(fn), write(fd), delete(fn)*, where *fn* is a filename and *fd* a system descriptor for an open file connection). These operations are classified into two categories: *i) Standalone*, which are based on the filename (e.g. *open()*), and *ii) Joined*, which are based on the descriptor of an open file connection (e.g. *write()*). The link over which a joined operation occurs, is established at run-time by a preceding standalone operation (e.g. an *open()* creates a link to a file that will be used by the *write()* operations that follow). Only standalone operations are using the naming mechanism of the file system. A file in the $\mathcal{FDFS}$ may reside at one or more LFSs. In response to a standalone operation, the interconnection environment tracks down and contacts the corresponding LFSs. If no corresponding LFS is found (i.e. file does not exist, or the corresponding LFSs do not provide the required properties), the interconnection environment raises an exception.

3 The $\mathcal{FDFS}$ in Aster

The $\mathcal{FDFS}$ goals pose various demands on the interconnection platform. Services provided by the base file systems should be available to the client, the interconnection system should promote sophisticated services on top of existing ones, and new services should be easily integrated. Aster enables the programmer to combine in a hierarchical manner a number of system components, to form a software bus customized to meet those demands.

3.1 Customization in Aster

An Aster application consists of three types of elements: *i)* the application modules which pose a number of requirements on their interactions, *ii)* the system level components which customize the interconnection environment according to the application requirements, and *iii)* the base-bus providing basic communication operations (i.e. *send* and *receive*). The application requirements are specified in terms of *property names* that are declared by the programmer in module interfaces. Property names represent properties recognized by the Aster environment, which are specified in an expandable property tree [1]. The tree provides the formal definition of each property by associating a logical formula to each property name. Based on the logical formulas, ASTER selects from the *system repository* an existing software bus and a number of system components, which can be combined to form the customized interconnection environment that meets application needs. Each system repository element provides a number of properties and may itself pose some requirements.

Let $\mathcal{B}$ and $\mathcal{S}$ be the set of software buses and system components in the system repository. We associate with each software bus $B \in \mathcal{B}$ the sets $\mathcal{R}_B$ and $\mathcal{P}_B$ which represent its requirements and provided properties. Likewise, we associate with each system component $S \in \mathcal{S}$ the sets $\mathcal{R}_S$ and $\mathcal{P}_S$. For an application A, with a set of requirements $\mathcal{R}_A$ and a set of provided properties $\mathcal{P}_A$, a software bus customized to meet its needs can be constructed if the following formula holds [1]:

$$\text{bus}(A) \equiv \ \exists B \in \mathcal{B}, \exists \{S_i\}_{i=1..n} \subset \mathcal{S} \ \ | \\ \{(\mathcal{P}_B \bowtie \{\mathcal{P}_{S_i}\}_{i=1..n}) \cup \mathcal{P}_A\} \lhd \{\mathcal{R}_A \cup \mathcal{R}_B \cup \{\mathcal{R}_{S_i}\}_{i=1..n}\}$$

where $\bowtie$ represents the operation of the *logic and* between each property of the selected software bus B with all the properties provided by the set of selected system components $\{S_i\}_{i=1..n}$ and the relation denoted as $\lhd$ states that when $X \lhd Y$ holds, the set of properties X guarantees *at least* the requirements of Y.

Based on that formula, ASTER can verify that a set of system components coupled with a software bus may satisfy application requirements, given that each module declares in its interface both the properties that it provides and it requires. In the remainder of this section, we use the global naming service to focus on module declaration in ASTER and on automatic construction of a customized software bus.

3.2 Customization for the $\mathcal{FDFS}$

The $\mathcal{FDFS}$ requires from the interconnection platform a mechanism, which locates the LFSs where a specified file resides. Figure 1 shows an ASTER software bus customized to provide such a location service through the use of a system component called *locator*. Each LFS enters the $\mathcal{FDFS}$ by registering its address to the locator. Standalone requests are received by the locator (step 1), which performs a search operation with parameters the interface type, the required properties and a selection function, in the set of known LFSs. When the proper LFS is tracked down, the locator forwards the request to it (step 2). Finally, the server returns to the client the result of the performed operation (step 3), establishing this way, the binding between the client and the LFS that satisfies the required properties. This type of binding is called *dynamic binding.*

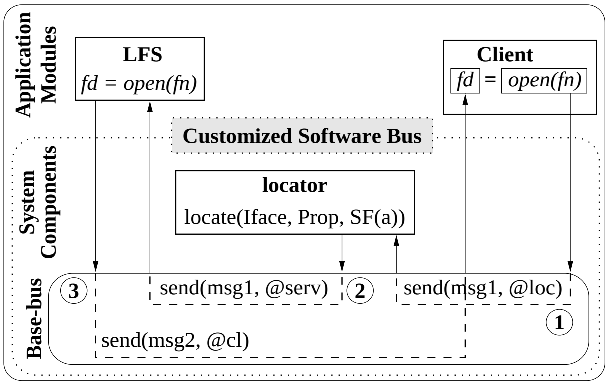

Figure 1: The software-bus during an open() request.

The *Iface* parameter in figure 1 specifies the interface type required by the client, and it is the trivial criterion in the search process. The *Prop* parameter specifies the required properties. For example, in order to open a file at an LFS, the client requires that the file resides at that LFS, hence the required property is *exist(filename) == TRUE*. The search among registered LFSs may not return a single answer (e.g. if the file is replicated). In such cases, results are filtered by a user-supplied selection function (*SF* parameter in figure 1) to return a single LFS. The selection function can be based on some *auxiliary* services provided by the LFSs (e.g. current load). If an LFS does not provide a required auxiliary service, a default value may be automatically assigned to it by the interconnection environment.

3.3 Describing the $\mathcal{FDFS}$ requirements in ASTER

Figure 2 contains a sample of interface declarations in the ASTER language. First, the client's interface is declared. The keyword *client* denotes that the specific module will issue requests for the declared operation. The declaration of the LFS's interface follows, and then a declaration specifying the file which contains the source code for a client. The fourth block contains the construction of the $\mathcal{FDFS}$ application based on the declared interfaces, and it is divided in three regions: *i)* the modules that participate, *ii)* the bindings between requests and services, and *iii)* the requirements for establishing those bindings.

```
interface clt-FDFS {
  client typeFD open(typeSTR fn)
  client typeBOOL exist(typeSTR fn)
  client typeINT write(typeFD fd, typeBUF buf)
}

interface LFS {
  typeFD open(typeSTR fn)
  typeBOOL exist(typeSTR fn)
  typeINT write(typeFD fd, typeBUF buf)
  typeINT load()
}

implementation ImplemClt-FDFS {
  client.c implement clt-FDFS
}

hierarchical FDFS {
  constituents ImplemClt-FDFS; LFS; selector;
  bind
    (ImplemClt-FDFS)open : (LFS)open;
    (ImplemClt-FDFS)exist : (LFS)exist;
    (ImplemClt-FDFS)write : (LFS)write;
  require
    (ImplemClt-FDFS)open(arg1) : dynamic(
      (LFS)exist(arg1), (selector)sf(): (LFS)load() );
}

interface I_locate {
  typeAddrSet locate(typeIF X, typeP Y)
  provides
    dynamic(X, Y);
  interconnection
    (client)send(C1, ∅, M=(X, Y, S, sf, msg)) ⇒
          receive(C1, my_id, M)
          s = (S)sf(locate(X, Y))
          send(C1, s, msg)
}
```

Figure 2: Declaring $\mathcal{FDFS}$ modules in ASTER.

The last block in figure 2 is the declaration of the locator's interface. It starts by declaring the offered services (i.e. locate), then specifies the provided properties (i.e. dynamic), and finally describes the procedure used for establishing the dynamic binding. Received messages are sent by clients who only know the type of interface of the server that they want to contact (parameter X), the type of basic services it should offer (parameter Y), and the application module (parameter S) which implements the user supplied selection function (parameter sf). The empty-set symbol $\emptyset$ denotes that the target LFS is not known to the client. Based on this information, locator tracks down possible LFSs and sends their addresses to the selector module, which takes the final decision and notifies locator. Finally, the locator forwards the client's original request to the selected LFS.

Based on the declarations of application interfaces (i.e. client and LFS), ASTER can *automatically* construct a run-time environment by re-using existing software (i.e locator), and verify its correct behavior as discussed in 3.1. The code for the interconnections among the application modules and the systems components is added by ASTER, leaving to the programmer only the responsibilities of precisely describing the properties required by the application. To provide the naming mechanism in $\mathcal{FDFS}$, ASTER chooses a system component that provides the following, *dynamic* property:

$$dynamic(\mathcal{X},\mathcal{Y}) \equiv \{IC : (IC \in \mathcal{I}) \wedge (type(IC) = \mathcal{X}) \wedge (\mathcal{Y}(IC) = \mathbf{TRUE})\}$$

where $\mathcal{X}$ is an interface type, $\mathcal{Y}$ is a specific property, and $\mathcal{I}$ is the set of all interfaces declared in the system. When an appropriate system component is selected, ASTER automatically configures the communication between it and the application modules, according to description of its interconnections as given in locator's interface (figure 2).

4 Related Work.

The changes done to the interconnection structure in order to achieve the required dynamic binding, form part of field called *dynamic reconfiguration* of a distributed application. This term refers to all kinds of run-time modifications on the interconnection structure of an application (e.g. migration of a module instance from one physical node to another, replacement of a module instance by another one, introduction of a new module, changes in the existing interconnections among modules, etc.), and is subdivided into two major categories: *i) programmed reconfiguration* (also seen as *ad hoc changes* [5] in bibliography), and *ii) evolutionary reconfiguration* (also seen as *post hoc changes* [5] in bibliography). The former category concerns modifications triggered by application generated events, and includes dynamic binding. In the remainder, the terms *dynamic binding* and *programmed reconfiguration* are used interchangeably. The latter category concerns modifications triggered by external factors (e.g. the replacement of a module by a more recent version, triggered by the administrator).

Most of the existing distributed configuration-based programming environments deal with dynamic reconfiguration, but not all of them have dynamic binding capabilities. CONIC [4] and POLYLITH [3] are two examples of such systems, which deal only with the evolutionary reconfiguration aspect. They provide primitives to intervene in the application execution and replace modules with their new versions or to add new modules to the application [11]. Using those primitives for achieving dynamic binding is a tedious job for the programmer, who is responsible to write the reconfiguration code and also provide a mechanism for its automatic engagement at execution time.

DARWIN [5], the evolution of CONIC, allows the programmer to explicitly declare operations authorized to access the existing interconnection configuration and modify it. By invoking those operations, a module is able to create/destroy an instance of another module, or to establish/destroy a link with another module. Similarly, in DURRA [12] the programmer can specify a number of alternative configurations for a given application, by describing the translations among possible configurations as result of execution events. In both cases, the programmer has to write only the conditions for the reconfiguration and not the whole reconfiguration code. However, the reconfiguration should be described in modules' code, which gives a static view on the set of possible changes in the interconnection structure of the application. Thus, for re-using those modules in another application with slightly different interaction schemes, the source code must be altered.

The UNICON system [13] provides a sophisticated model for describing the relations among application modules and specifying the set of operations allowed to issue on modules interconnections. In UNICON terminology, locator can be viewed as a *connector*. Given the application components that should be bound and a connector that can be used for this binding, UNICON verifies that it is the appropriate connector and establishes the interconnection channel. However, it does not reason on the application requirements, in order to provide automatic selection of the appropriate connector from a set of pre-existing modules. Thus, the programmer rests in charge to provide all the modules that participate in a binding.

CORBA [6] provides dynamic binding capabilities through the use of the *Dynamic Invocation Interface* (DII) that accesses the *interface repository*. It also provides a system component named *trader*, which is able to contact any application module during execution time and modify its links with the interconnection environment. Thus, combining the interface repository with the trader suffices to provide programmed reconfiguration capabilities. The difference with ASTER, is in terms of facility and effectiveness on programming them and not in terms of final result. In CORBA, the programmer writes the code for establishing the communication between the trader and the interface repository whereas in ASTER, the programmer only specifies the dynamic binding requirements and the system establishes the appropriate connection automatically.

5 Summary

The $\mathcal{FDFS}$ is an approach for using a distributed programming environment to construct the interconnection platform for integrating different existing file systems into a single one. The idea is to provide the user the whole range of facilities that are implemented by the various existing file systems. As a first step in this effort, we demonstrated how to provide the user with transparent file access, by describing the construction of the $\mathcal{FDFS}$ naming mechanism.

Future Work. Besides the name service which is a basic step in the design of a file system, today's file systems offer a wide range of services which make them attractive to the users. We envision that the $\mathcal{FDFS}$ based on ASTER will be able to integrate all those services in a robust system, and also give to system administrators the ability to create complex services as a combination of basic LFS services, by constructing appropriate interconnection structures customized to meet the new requirements. The hierarchical structure of ASTER interconnection environment allows us to integrate ideas from the layered file system architectures [14], so as to efficiently combine contrasting services (e.g. access to secure data with fast block-data transfer over a secure channel provided by a customized interconnection environment). Moreover, the $\mathcal{FDFS}$ could combine a number of LFSs which implement simple replication policies, to build a complex system corresponding to a sophisticated availability model required by a client.

Conclusions. The global naming mechanism for the $\mathcal{FDFS}$ is shown to demand great flexibility in dynamic binding from the interconnection platform. We have demonstrated that ASTER fulfills those demands and facilitates the programmer's effort to build an interconnection platform sufficient to meet the $\mathcal{FDFS}$ needs. Comparing ASTER with other distributed programming environments, one can see that our system is better suited for implementing the naming mechanism for the $\mathcal{FDFS}$. We believe that ASTER can be successfully used for gradually adding services to the interconnection platform so as to provide a wide spectrum of file system facilities, ranging from simple ones (e.g. providing secure access to the data of a given file by combining LFS encryption facilities and a customized interconnection environment providing secure channels) to really sophisticated facilities (e.g. system components implementing algorithms for automatically recognizing files with high demands for concurrent access and migrating them to LFSs that provide parallel file access). $\mathcal{FDFS}$ is a potentially open-ended system, able to accommodate LFSs providing properties not yet known, based on ASTER's truly dynamic binding mechanism.

More information on ASTER work is available at: http://www.irisa.fr/solidor/work/aster.html

Aknowledgements

Jean-Pierre Banâtre provided insightful comments and significantly contributed to the perception of the $\mathcal{FDFS}$ from the ASTER point of view.

References

[1] V. Issarny, and C. Bidan, ASTER: A Framework for Sound Customization of Distributed Runtime Systems, *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[2] V. Issarny, and C. Bidan, ASTER: A Corba-based Software Interconnection System Supporting Distributed System Customization, *Proceedings of the International Conference on Configurable Distributed Systems*, May 1996.

[3] J. Purtilo, The POLYLITHSoftware Bus, *ACM Transactions on Programming Languages and Systems*, 16(1), 1994.

[4] J. Magee, *et al.* Constructing Distributed Systems in CONIC, *IEEE Transaction on Software Engineering*, Vol. 15, June 1989

[5] J. Magee, *et al.* Structuring Parallel and Distributed Programs, *Proceedings of the International Workshop on Configurable Distributed Systems*, March 1992.

[6] OMG Object Management Group, The Common Object Request Broker: Architecture and Specification - Version 2.0, *OMG Technical Report*, July 1995.

[7] S. J. LoVerso, *et al.* sfs: A Parallel File System for the CM-5, *Proceedings of the summer USENIX Conference*, June 1993.

[8] Y. A. Khalidi, and M. N. Nelson, Extensible File Systems in Spring, *Proceedings of the ACM Symposium on Operating Systems Principles*, 1993.

[9] A. D. Birrell, *et al.* The Echo Distributed File System, *DEC - Systems Research Center Technical Report SRC-111*, 1993.

[10] M. Satyanarayanan, *et al.* The ITC Distributed File System: Principles and Design, *ACM Operating System Review*, 19(5), 1985.

[11] J. Kramer, and J. Magee, The Evolving Philosophers Problem: Dynamic Change Management, *IEEE Transactions on Software Engineering*, Vol. 16, November 1990.

[12] M. Barbacci, *et al.* DURRA: A Structure Description Language for Developing Distributed Applications, *Software Engineering Journal*, March 1993.

[13] M. Shaw, *et al.* Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[14] R. G. Guy, *et al.* Implementation of the Ficus Replicated File System, *Proceedings of the summer USENIX Conference*, June 1990.