Client Side Reconfiguration on Software Components for Load Balancing

Erik Putrycz and Guy Bernard
Institut National des Télécommunications
9, rue Charles Fourier
91011 EVRY Cedex, France
{*erik.putrycz, guy.bernard*}@*int-evry.fr*

Abstract

The size of applications is becoming greater every day. Software components enable developers for easily building large applications with basic reusable parts. However, there is a need for a system support to allow these applications to scale to many users and to large networks. Current methods for load balancing cannot be applied, since they are mostly designed for local area networks and the monitoring tools are not sufficient. To allow efficient load balancing, we first have to consider the architecture of the monitoring infrastructure. In this paper we present a way to build a scalable monitoring platform. By using information provided by this platform, it's possible to build policies for helping system administrators in the management of complex tasks. Using application descriptions, these policies take into account the global architecture and drive reconfigurations using the monitoring data.

1. Introduction

Today, with the increase of the size of applications, reusable software components bring a new solution to lower development time and cost. Moreover, software architectures - components connected together on different hosts - allow to build easily large scale distributed services, but new criterias have to be considered for designing such services:

- applications are more and more distributed on large scale networks due to the distribution of resources and tiers;
- the user is asking for a better system support to have a better experience.

Static solutions for those problems are no longer possible because of the dynamic constraints of those applications: while an application is running, the network performance and the state of execution environments can significantly change. It's hard to predict even on short term the behavior of software parts execution. There is a need of a dynamic system support for adapting those applications to their environment; for applications built on distributed software components, this means that their execution platform has to be fully re-configurable. Reconfigurations can be either functional or non functional. Functional reconfigurations are application-driven in order make structural changes. In the other hand, non functional reconfigurations should be transparent to the application developer. They can be used for load balancing or fault tolerance.

Jim Gray said: "The load balancing problem is easy to describe and hard to solve". In fact, it can be described in a few questions: Given a number of requests for a host, how should this host react ? Should it handle all incoming requests itself, or should it transfer some requests to another host ? If yes, how to choose this host ? And what is the cost of this mechanism ?

The first requirement for load balancing is a distributed monitoring platform. Most monitoring models are based on a client server model. This means that when the client needs to get the state of another host it has to query it with a network protocol. With low latency, it is acceptable to take blocking decisions on this data. But as the latency gets higher and the number of hosts increases, this decision time will be higher. Thus, this model can hardly scale to large networks. For making a load balancing service scalable, information has to be distributed the same way as the decisional infrastructure. The main goals to achieve are:

- precision: the quality of decisions depends of the precision of the information collected;
- ubiquity: the information has to be accessible from any host;
- non-intrusiveness: monitoring has to interfere as little as possible with application execution.

Heisenberg's Principle states that one cannot measure something without also affecting it in some way. This ap-

plies to monitoring too and so the main challenge is to find a good trade-off between the three characteristics listed above. Moreover, those three parameters are linked: an increase of precision or ubiquity induces more network traffic and is therefore more intrusive.

This infrastructure is first and foremost of benefit to the system administrator. But managing a large application by hand is a hard job. To assist him/her, reconfiguration policies are designed for complicated tasks like load balancing. Reconfiguration policies base their decisions on monitored data and act according to the reconfiguration capabilities of the underlying system. In the recent years, work on load balancing policies has been mostly applied to processes or databases, and the decisions of load balancing policies are generally based solely on the CPU load of hosts or a random algorithm. With software components, dependencies between components are explicit and this allows a policy to manage a whole application infrastructure.

First we will study the needs at the monitoring level and present our implementation (Section 2). Then, we present all the reconfiguration methods which are the basic tools for load balancing (Section 3). With precise monitoring data and a high level tool we are then able to define policies (Section 4).

2 Passive distributed monitoring

We define distributed monitoring as the gathering of significant data describing the state of a system. The system represents the global service or application that we want to monitor. The data can be characterized by all information that has a role in the execution of this system. The state is the information at a fixed time *t*. Significant means that the state changes have to be minimized to reduce the intrusiveness side of monitoring. First we will study what to monitor and then we present our implementation.

With software components, there are two levels to monitor: the application level (the data flow between components), and the state of the hosts on which components are executed.

2.1. Monitoring an application behavior

There are many different ways to monitor an application based on software components. To clearly understand the needs we first have to describe the different communication levels (Figure 1). A software component is executed inside a component runtime, this runtime handles all non functional properties (e.g.: transactions for Enterprise Javabeans [12] and COM+ [4], synchronization and threading in COM+). This component runtime needs to access high level communications to manage those non functional properties and to allow communications between components.

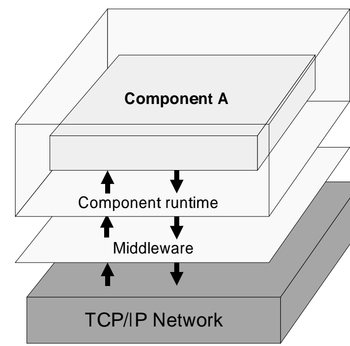

Figure 1. Different communication levels

The layer responsible for this task is usually a middleware (RMI for Enterprise Javabeans, DCOM for COM+ components, CORBA ORB for CCM[7]). Monitoring the middleware can be done by two ways:

- interception: the middleware data frames are captured and re-decoded to analyze the content;
- integration: hooks are added inside the code of the middleware itself to inform the monitoring runtime of some events.

Most work in this field has been done on CORBA based middlewares. Goodewatch [3] is based on the integration approach: hooks are added into the ORB to intercept *invoke* calls. Eternal [6] catches CORBA calls at the TCP/IP level using system call interception. The interception approach allows to catch middleware method calls without any modification of the source code itself but as it relies on low level network functions, it depends on operating system interfaces.

Getting information at the middleware level is not necessary for our needs. Because our basic re-configurable unit is a component (Section 3), information gathered from other levels than components is not very useful for reconfiguration policies. So we have chosen to capture only communication between components without runtime calls for additional non functional properties (transactions, etc.). This can be done by watching communication at the runtime level.

2.2. Host level information

Monitoring has to take the core computing capabilities into account. These are related to the state of each host. To explain the influence this state has on an application, we focus on the roles of processes and the network.

2.2.1. Evaluating network cost. In the case of our target application, communication between components usually consist of short requests and short answers. Such communication based on middleware is very often synchronous and a client is blocked while waiting for an answer from a server. This is why the network latency between hosts is the most important parameter especially in wide area networks like the Internet ([2]). This factor is furthermore increased by the fact that remote invocations with middleware usually induces many TCP/IP messages (with CORBA and RMI, our tests showed that one two-way invocation needs 4 messages). So the most important information to capture on the network layer is this latency delay.

2.2.2. Role of processes. As described in Figure 2, the mapping of components to processes is decomposed into three levels. The application consists mainly of dependencies and communication descriptions between components. At a host, a component will be running inside a process (depending on the component platform, different mappings to processes are possible). Process monitoring is a well studied topic. Most operating systems have many tools for analyzing performance. The main parameter is CPU use. The second one is memory use. Understanding the memory use impact on execution is not easy. Most tools show only the memory available but this has nearly no meaning, as most operating systems (Linux, Solaris, Windows NT) will use all available memory for file caching purposes. This memory can be released on demand. And for extending memory beyond physical memory capacity a well known mechanism is virtual memory. Virtual memory is stored on a disk and so has a nearly 100 fold slower access time. As a result, applications using virtual memory are hugely slowed.

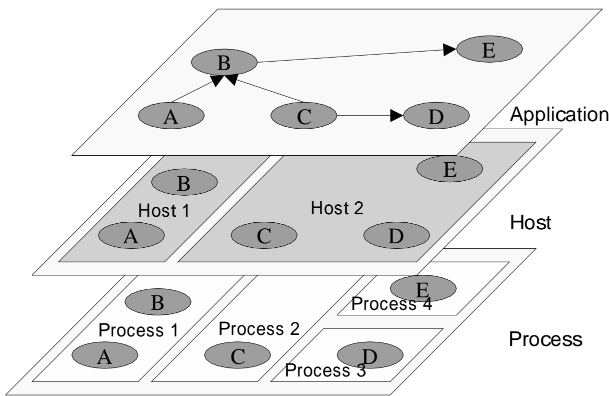

Figure 2. Different abstraction levels

Methods for allocating virtual memory differ a lot from one operating system to another. The most known are *paging* (BSD like, Solaris, Windows NT) and *swapping* (Linux). Paging is performed continuously by a system daemon which stores all unused memory on disk. As consequence, the paging rate is the indicator of memory shortage [1]. Swapping on the other hand is activated when there is no more free memory available. Every new memory allocation will be done on a swap disk space. The percentage swap space used in this case is the main indicator.

2.3. Design

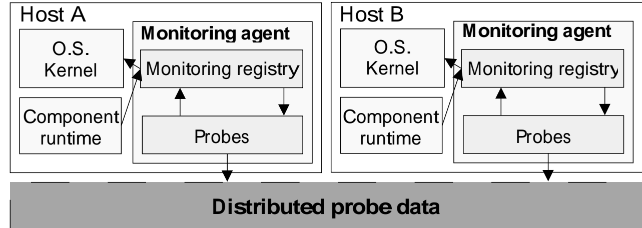

Figure 3. Distributed monitoring architecture

To fully understand the needs of our structure, we need to explain issues with current monitoring systems. The usual way to collect information for monitoring is to get a fixed intervals some data from hosts (*polling*). The most common implementation is SNMP (Simple Network Management Protocol). An agent at each host responds to queries for basic SNMP objects. But this approach cannot scale (without an important data overhead) and is not precise. It is based on the client-server paradigm, and this is not suitable for reducing intrusiveness: the quantity of network traffic induced depends linearly of the precision of data. To solve this problem, our implementation is built on two levels of information. The first level is called the monitoring registry. This level is used for local storage of low level information (Figure 3). Those data can be updated either in-process for kernel information at a fixed rate (CPU load, virtual memory use, etc.), or out-process by events (e.g. a component invocation). In order to keep this information flow (polling or events), there are events from *monitoring registry* to *Probes* data.

In our prototype based on Sun's Enterprise Javabeans Reference implementation, we added calls inside EJB InvocationManager *preinvoke* method which informs our monitoring agent. This allows us to get statistics on the data flow between components. This is used for reconfiguration policies (as explained in Section 3) and for administrative needs.

The entity responsible for interpreting and filtering *monitoring registry* data is called a *Probe*. It has access to a second level of data (*Probe StatValues*) which is distributed to all hosts of the monitoring federation. Updates of this data should only be done when significant changes occur (this is the role of the filter) to minimize remote updates. This data

is platform independent for administration and reconfiguration policies usage: e.g. for CPU load, the data represents a percentage of utilization and is updated on detection of a 10% average change every 10s. Using filters and events allows the client to have a passive role in monitoring and so it will be easier for it to adapt only on important environment changes. Those adaptations are based on policies. The possibilities of these policies will depend on dynamic reconfiguration capabilities.

3. Dynamic reconfiguration

Future distributed applications will need to be able to easily scale to heavy loads. Dynamic reconfiguration tools allow evolutions without degradation of service. There are mostly two goals:

- non functional reconfigurations: they have to be completely transparent to the final developer. They have not only to change component locations but naming references and clients bindings too.
- functional reconfigurations: they are mostly for structural changes in the application. This kind of reconfiguration is upon the naming references and provides only for the application developer an API to a reconfiguration tool.

3.1. Migration

The migration mechanism is a transfer from one host to another of an entity. There are two sub-types of migration: strong migration with a transfer of the state of this entity, and weak migration with only a transfer of the executable part and some attached data. This mechanism should have a strong property of atomicity and consistency. Migration should be seen as one operation which can restore the previous state in case of failure (atomicity). Consistency means that the execution flow of the application should remain unchanged after the migration of a component.

3.2. Replication

Replication is a well studied topic in database research. Its goal is to increase the locality of reference [9]. The main idea is that by duplicating components at different sites it is possible to enhance performance, reliability and availability. For example if a host is performing many accesses on a remote component there may be a worthwhile performance gain by duplicating this component locally. This helps to solve some reliability problems too: if one host fails the component will still be accessible somewhere else.

With databases, replication can be performed by copying data to another host. With components there is not only a data part but an executable part too. This executable part can be divided into a binary part and its state. So the replication mechanism should take care of duplicating them on other hosts.

It's possible to distinguish two different types of replication :

- load balancing oriented: its goal is only to enhance performance in choosing the right host to execute every divisible part (method in stateless component or session for stateful components).
- fault tolerance oriented: for enhancing reliability components are duplicated. A host has to manage the state of every copy and this constraint usually decreases performance. The state of every copy can be managed by two ways :
  - active replication: the state of each replica is maintained by broadcasting each query to each host (1 to N scheme);
  - passive replication: only one replica receives all requests and a subsidiary mechanism is responsible for duplicating its state (1 to 1).

Handling the state of an executable is not an easy task because native functions of most languages like Java or C++ don't allow serialization of threads (the basic unit of execution). This is why quite often a deteriorated form of replication is used: the state of the component is not transferred and the replication process is executed only when the component is in an idle state (no sessions are opened by users).

3.3. System tuning

A lot of system parameters can be adapted to meet the need of a specific application. Those parameters range from operating system level (disk access methods, different OS cache sizes), to resources (e.g. databases) and to the component runtime. Some of these parameters are fixed at startup but others are dynamically modifiable. Thread usage is one of the most important parameter for performance on the component runtime. Allowing many components to execute, each one in its own thread, can improve dramatically their individual execution times: on multiprocessor systems the system scheduler can dispatch threads on all processors, so many methods can be executed at the same time. But allowing too many threads to execute at the same time would be costly. Each thread will need some memory, and scheduling time would be increased. That's why the most common approach is to have a fixed pool of threads (this is implemented in most component servers). The number of running threads is an interesting parameter to change under heavy load for avoiding resource overload.

3.4. Constraints

Reconfiguration of components is possible only under some conditions. To achieve a particular behavior or function in a framework, a component may be bound to specific resources. This induces localization constraints on some parts of the application. What we mean by bound resource is that communication between the component and the resource can't be done remotely (because of native system calls or of network cost which would unacceptably reduce performance). As an example, take a graphical user interface component (GUI), which generates lots of communications (usually native calls) with the operating system. This component can't be executed somewhere else. And the GUI will be at only one user side so it can't be duplicated.

Another constraint can be a specific co-localization need of several components. Some components have to be kept in the same memory space than others to avoid heavy remote interactions. For instance, a web browser component can be generic enough to take an URL and render it into some graphical format (it will not have any resource binding). Such a component has to be co-localized with its control GUI (which is bound to a graphical display) to ensure low cost inter-process communications between those two components.

For these reasons, some components are not appropriate for reconfiguration.

4. Reconfiguration architecture

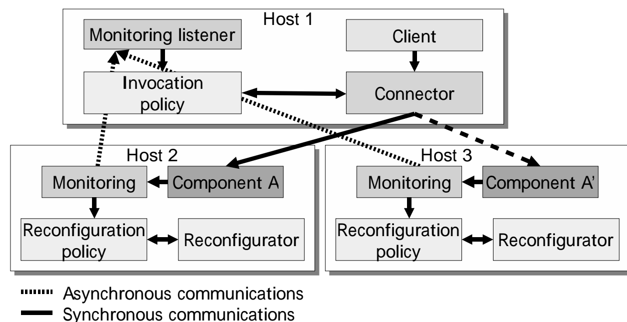

Figure 4. Reconfiguration framework

In order to create a load balancing service, we have build a distributed reconfiguration tool. Our model is described by an example in Figure 4 : Host 2 is hosting a component which has been replicated to Host 3. The client is the program part running on Host 1 which is accessing remotely component A. Host 2 and Host 3 are sending monitoring events (section 2) to the client (Host 1). Using this information, the client is able to reconfigure itself based on an invocation policy. The *Connector* is our interface to change the client bindings. The key to reconfiguration is that every used component is associated with a *Connector* : the *Connector* contains informations about the binding and the replication pool of a component. On the server side the dynamics are handled by a reconfiguration policy (section 4.2).

4.1. Client side dynamics

The first goal is to be able to rebind the remote reference of the client to another server. This remote reference is handled by a stub (on nearly every middleware platform). The stub then issues *invoke* calls on the ORB (Object Request Broker). Precisely on CORBA (or RMI over IIOP for Enterprise JavaBeans) those calls are handled by the *org.omg.CORBA.portable.ObjectImpl* which transfers the calls to the vendor implementation. We have inserted a reconfigurable connector inside this method to change the real remote reference. By this way we provide a *rebind* method able to transparently execute remote methods on any component.

The second important point with this kind of reconfiguration is to manage the references. On Enterprise JavaBeans and CORBA Component Model a *Home* object is responsible for locating and creating objects on the server side. We intercept those calls too to associate a reference with a component name and a remote host. Actually we suppose that the remote components are stateless and every invoke call can be executed independently on any server. If a state is involved the *rebind* method has to take care of the sessions bound to the component state.

Figure 5 shows a sample use of the reconfiguration interface and the monitoring tool to build a load balancing invocation policy. We suppose the client is using a component A which is replicated on some hosts. When the client receives an update of CPU load of those hosts (*OnStatEvent*) it will rebind the Stub to the replicate on the less loaded server.

4.2. Server side dynamics

The server side needs to evolve to adapt the component hosts to their environment. This can be achieved in adding a reconfiguration interface to every server. This interface is able to install new components and manage instances in order to replicate or migrate components. It interacts remotely with the clients *Connectors* to update their replicated pool and references.

Reconfiguration can be driven for resource use or availability. For instance a resource based policy may want to migrate a component to a remote host to free resources. If there are enough resources available a replication policy could replicate a component on many hosts to increase its

```
public class LBInvocationPolicy
       extends InvocationPolicy {
  ...
  // on receive of monitoring data
  public void OnStatEvent(StatEvent stev) {
    Connector c = cm.getConnector(clientRef);
    // get all the replicated components
    ComponentRef[] rp = c.getReplicationPool();
    InetAddress hostname =
                clientRef.getComponentHostAddr();
    ComponentRef newRef = clientRef;
    long min_load = sm.getStat(hostname,CPU_VALUEID);
    // get the minimum load value from all
    // hosts running a replicated component
    for (int il = 0;il < rp.length;il++) {
      InetAddress replhost =
                rp[il].getComponentHostAddr();
      repl_load = sm.getStat(replhost,CPU_VALUEID);
      if (repl_load < min_load) {
        min_load = repl_load;
        newRef = rp[il];}}
    if (newRef != clientRef) {
      // rebind the client to the new host
      c.rebind(newRef);}}
}
```

Figure 5. Sample of a load based invocation policy

availability. Another way to improve the global execution is to migrate components in order that two components that have many interactions are executed on the same host. All those different policies can be build by different providers implied in distributed software services : for example the resource based policies are very dependent of OS specific data and thus should be build by the monitoring tool provider.

5. Related work

Most of the work on load balancing at the middleware level is limited to load balancing for CORBA objects on local networks. Usually they are based on a *Delegator* pattern [10] on the server side: one host is receiving requests and it intercepts and delegates them to another server. This host manages a pool of replicated CORBA servers and chooses at each request one of those replicates. Most policies for this choice are based on round robin or random algorithm [5, 11]. Such policies don't take into account the application architecture and the state of the replicates.

The most complete work on load balancing for CORBA is the Load Balancing Service of TAO [8]. Its architecture is quite similar to our model: monitoring can be synchronous (pull) or asynchronous (push) and it allows adaptative policies for load balancing too.

For scaling load balancing to large networks, the main problem is to reduce all interactions. We solved this problem with a fully decentralized reconfiguration architecture and smart filters for monitoring data.

6. Conclusions and future work

In this paper we have presented our approach for building a generic load balancing service for application based on software components. In order to be able to make good decisions, this kind of service has to rely on precise monitoring data. The policies are fully distributed which has consequences in the whole infrastructure. By distributing only significant data, the monitoring service is not overly intrusive and allows to easily react to those changes. Using this data, we have a model which involves all the actors of an application in load balancing policies. Implying many providers means that the reconfiguration platform should provide a high level of pluggability on every part (policies, monitoring tools). This can be achieved in conceiving the whole platform as a framework.

Parts of the platform are still under development and we expect to have soon real performance results. There can be a multi-criteria problem to solve for some policies so we are working on methods to compare the cost of each parameter. We are also investigating how to replicate any kind of component, even one with a persistent state.

References

[1] A. Cockcroft and R. Pettit. *Sun performance and Tuning*. Prentice Hall, 1998.

[2] P. Ferguson and G. Huston. Quality of Service in the Internet: Fact, Fiction or Compromise ? *INET'98*, July 1998.

[3] C. Gransart, P. Merle, and J. Geib. Goodewatch: Supervision of corba applications. In *ECOOP'99 Workshop*, Lisbon, Portugal, June 1999.

[4] Microsoft. Microsoft developer network. http://msdn.microsoft.com, 2001.

[5] W. Monin, B. Nicolas, and Y. Gourhant. Performance evaluation by simulation of otm load balancing systems. In *Proceedings of NOTERE 2000*, Paris, France, Nov. 2000. in French.

[6] P. Narasimhan, L. Moser, and M.-S. P. Using interceptors to enhance corba. *Computer*, July 1999.

[7] Object Management Group. *Corba Component Model Specifications*, July 1999.

[8] O. Othman, C. O'Ryan, and D. Schmidt. The design and performance of an adaptive corba load balancing service. *Distributed Systems Engineering Journal*, 2001. to appear.

[9] T. Özsu and P. Valduriez. *Principles of distributed databases systems*. Prentice Hall, second edition, 1998.

[10] D. Schmidt, M. Stal, H. Rohnert, and F. Bushmann. *Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture*. Wiley, 2000.

[11] T. Schnekenburger. Automatic load distribution for corba applications. In *Proceedings of CUC Component User's Conference*, Munich, Germany, 1996.

[12] B. Shanon. *Java 2 Platform Enterprise Edition Specifications, v1.2*. Sun Microsystems, 1999.