A Design for JTrader, an Internet Trading Federation

Marcelo d'Amorim Carlos Ferraz

{mbd, cagf}@cin.ufpe.br
Universidade Federal de Pernambuco
Centro de Informática
Caixa Postal 7851, 50640-970, Recife-PE, Brazil

Abstract

By using templates for service searching, Service Discovery Protocols (SDP) enable automatic discovery of network components, meaning less administration for the user and service owner achieved by a plug-and-play semantics. Moreover, the reliability of the trading system, including the service offers it contains, is well handled because of availability is a common concern. Despite the fact that these protocols were first concerned with connecting devices in a local scale network, they also play an important role on information service trading which is very relevant considering the Internet's large scale. This work illustrates SDP scenarios and motivation, briefly introduces current technology and finally presents the design of JTrader, an Internet Trading Federation based on Jini Technology. This system main purpose is to address root problems around service trading on the Internet, like security and availability, providing a federation whereby services could be globally localized and used.

Resumo

Protocolos de descoberta de serviços como SLP, Jini, UPnP e Bluetooth têm demandado grande interesse na comunidade científica e na indústria, uma vez que tais protocolos dão suporte a configuração dinâmica de aplicações. Usando características dos serviços, esses protocolos, conhecidos como SDP (*Service Discovery Protocols*), permitem a descoberta automática de componentes de software em uma rede. Isto significa menos esforço de administração para o usuário e para o administrador, alcançado através de uma semântica *plug-and-play*. Apesar destes protocolos terem sido concebidos para conectar dispositivos eletrônicos, eles apresentam um importante papel para a realização de *trading* de sistemas de informação, o que é relevante considerando a larga escala de uma rede como a Internet. Este trabalho apresenta motivação e alguns cenários de uso da tecnologia SDP, faz uma breve introdução à tecnologia atual de trading, e finalmente apresenta o projeto do JTrader, uma federação de serviços na Internet que utiliza Jini como ferramenta.

Palavras-chave: Jini; Internet: protocolos, serviços e aplicações.

1 Introduction

Currenlty, mobile communication is being considered a very promising market due to the rapid popularization of cellular telephones and devices such as laptops and PDAs. In the near future, it is expected a strong deployment of services in order to fulfill this demand. Location transparency is not a recent problem but has assumed great importance in an environment where clients change their place very often. With service discovery protocols, devices (or simply clients) find network services by its properties and services advertise their availability in a dynamic way. On the other hand, assuming the Web moves from document to program containment, this network will be the greatest distributed system ever seen with numerous programs for all kinds of tasks. This diversity implies a greater number of bindings among programs. To guarantee system's bindings stability, components must be super-reliable, which is very hard to achieve. The alternative is to build distributed applications layered over an intelligent service discovery framework whereby the bindings between components can be (re)discovered at runtime, and do not have to be static [12] - this is called *service trading*.

This work presents a design for an Internet service federation which help users find and use global services based on its characteristics. As we will see, enterprise components wishing to deploy their information services over the Web should notify their interest to the global trading system by means of Internet agents. These distributed components serve as monitors, reporting to the centralized federation every relevant event like a network failure or some modification on a remote service state. Section 2 presents some trading scenarios and segments their use in two major blocks. The current main technology for service trading is presented in section 3. Section 4 presents the design of JTrader, a Jini-based service discovery system over the Internet. Finally, section 5 concludes this work.

2 Service Discovery Scenarios

Connecting services into a network and discovering these services *on-the-fly* by means of its characteristics form the basis of trading systems. This works discusses the use of these kind of systems over the Internet where new services are installed every day. Service trading seems to be very appropriate to this dynamic Internet since it is based on an asynchronous and very decoupled distributed programming model that allows client to discover services with very few information about it. For instance, there is no implicit order for starting components: first the server; next, the clients, which would mean a difficult item of configuration, considering the Internet large scale.

On this approach, services are grouped on communities and these communities can be federated thus growing on scale; so, users could find services by means of its characteristics and would access them by its programming interface or using directly its graphical interface, if provided. Discovering communities, joining services on these communities, and searching for services are traders key responsibilities. Next session presents current trading technology where these tasks are better detailed.

At a first glance, traders could be considered a sophisticated name server as long as it does not need logical names to find distributed components, but their characteristics: service type and properties, for example. On the other hand, traders present some features other than location transparency:

- *Software Integration (Configuration).* Discovering and binding to services without using static and non reliable information enable unprecedent opportunity to seamlessly software integration and dynamic configuration in unknown environments.

- *Availability and fault-tolerance.* Given the Internet large scale and dynamism it is hoped that components of such systems could negotiate on presence of failures in a way that only running services could be accessed, services and communities find each other when they start up, and service crashes occur in a smooth fashion.

- *Service Independence and Interoperability.* Service is the most important concept for traders. It is an abstraction for a general capability. No matter if it is a device command interface or a simple e-mail API, traders must expose them as an interface by which users can access. Considering the interface the service exposes is language neutral, interoperability with heterogeneous programming languages could be achieved seamlessly. But interoperability does not relate only to language, but also to communication protocols used.

- *Simplicity.* The programming model must also be very simple to enable the engagement of new programmers on this network and also enable dealing with distributed programming issues.

- *"True" Location Transparency.* To assure this property, not only the service but also communities must be located without physical location information. There are some alternatives to searching for services on traders other than those based on names and URLs (*Uniform Resource Locators*) that we currently use.

User Interface is also a very important issue assuming there are environments with different constraints trying to access services. Programs searching for services in traders provide, in general, their own user interface and access programmatically the public service API, so the service user interface is not a concern. In contrast, when users access the trader, they mostly search for some terminal service - a service that provides a complete user's functionality - and expect to interact with it by means of some particular user interface suitable to the available device used. Figure 1 illustrates traders different kinds of use.

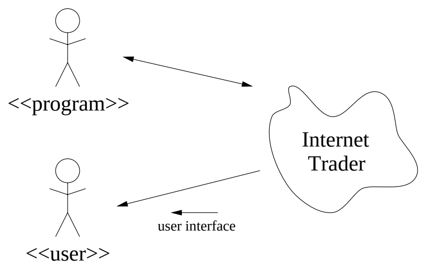

Figure 1: Ways to access a trader

Although we describe these properties as trader responsibilities, not every trader implementation provide all of them. Also, some properties, like fault-tolerance, are not achieved only by the middleware used but by a service, client, and trader responsibility contract.

This work makes a subtle but very relevant distinction between trader and service trading. The last represents a broad semantics for distributed computing where a programming model lies on. The former represents a tool by which trading is achieved. For example, when using CORBA for trading, it is likely to use a subset of services - besides the Trader Service - in order to provide a complete solution for distributed computing with trading semantics. Despite this difference, from now, this article use interchangeably both words. The distinction between both meanings depends on the context.

The opportunity to discover services on demand makes traders distinct from other types of distributed programming model since the dynamic nature must be considered at design time. From now, it is discussed what kind of systems traders fit best.

2.1 Device Oriented Applications

As long as clients access services by an interface, there is no assumption on service implementation. In fact, devices must provide some support for trading, but they do not necessarily require embedded applications. For example, a micro-wave provides a serial interface by which it can talk to the external world; in this case, some wrapper service implementing the micro wave common interface[1] can be defined to access the serial interface thus, since this service joins some client accessible community, the micro wave oven can be commanded remotely.

There are several patterns of communication for device oriented applications. Some patterns are likely to be more adjusted than others, depending on device's constraints and application purpose. For example, to enable service trading on devices, the direct solution is to install a runtime environment with support to the distribution platform, but sometimes, these constraints do not apply to devices with lack of memory and processing capacity. With the popularization of mobile technology some trading scenarios can be envisioned to the near future. For example, a street walker - owner of a capable mobile telephone - could be notified to special products offers when passing in front of a store. Getting into the store, he could search for some particular product by browsing the store's service through the mobile device.

2.2 Information Systems

Even with the current Internet, most software arrives at companies by means of specialized software revenue firms. As Internet is growing and creating opportunities, new software commercialization approaches starts to be investigated as long as software turns into a commodity. This means big changes in the way you acquire technology, and in the way your company uses technology for commerce. Also, even bigger changes must be considered in the way software is actually made in order to provide scale to this commercialization. Renting applications is a compelling option for many companies. For every application you offload, it means at least one less software system to purchase or license, maintain, and support. It also means deployment is faster [7]. Although traders were not designed concerning this use, its dynamics fits well to this kind of software commercialization as client and services are not statically bound. For example, some auctions systems and e-commerce marketplaces products could already benefit of its characteristics, as discussed in [7].

Traders enable high availability and fault-tolerance of services as long as there is no static association between services and clients what means a client could bind to a different implementation after a service crash. However, to achieve this in some systems, the state should also be very resilient and decoupled from the service, so another active implementation

[1]Industry leaders are defining standard interfaces to worldwide used devices like VCR, printers and microwave ovens.

could catch that state and continue servicing after a failure. Implementing service persistence on databases or non-transient blackboard systems like [11] and [13] can help to deal with this issue.

3 General SDP Architectures

This section does not intend to present a tutorial on SDP technology, but defines a structural model on which these protocols are based, thus generalizing the solution as long as it is presented at a high abstraction level.

Analyzing trader solutions under a structural point of view, we identify three basic components: directory agents (DA), user agents (UA) and service agents (SA). The DA component is responsible to store service offers and execute queries by matching the stored offers against query constraints, which can result in a set of service items. A service offer describes an available service implementation and can aggregate some important properties like the service location, for example. In addition, a service offer provides a means to access the service implementation it describes. The offer should store a remote reference to an object, like CORBA [10] does; an IP address and port where the service is running, like SLP (Service Location Protocol) [8] does; or a full-fledged proxy object that directly or indirectly implements the service interface, like Jini [1] does.

UA components are responsible to elaborate queries on behalf of users and submit these queries to the trader which may answer with a non-deterministic [2] set of service items that satisfy query constraints; so, it is up to this component to select the best item(s) returned. On the other hand, while a service implementation keeps running, the SA component is in charge to maintain service offers consistent on DAs. Figure 2.a shows the relationship among these components. In order to perform these tasks, user and service agents need to find a suitable DA to, respectively, search for service instances and keep available service offers. In fact, a *discovery* protocol enable these components to find directory agent instances.

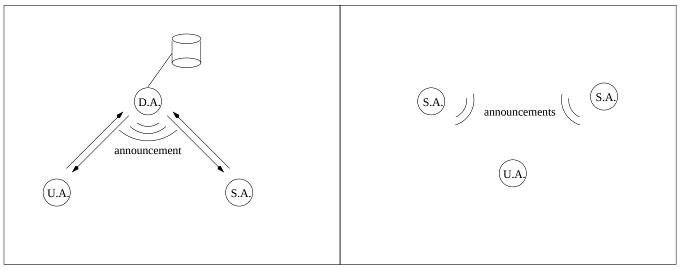

Figure 2: Trading architectures - (a) DA-based and (b) peer-to-peer approaches

The most common used discovery protocol version makes use of unicast communication; however, it can leads to lack of location transparency as UA and SA components are statically bound to a physical address where the DA instance hosts. As a way to achieve location transparency by not addressing messages to some specific machine and port, some SDP protocols implement discovery using multicast communication. Actually, multicast discovery protocols can be active or passive. In the first case, a user agent start the protocol sending a request message to some known multicast endpoint and receive callback messages that

[2]Depending on the query method used. Some methods are synchronous and may impose timeout boundaries.

running DAs, which listens to that endpoint, send. System administrators very often define groups for DA's participation as a means to segregate service categories. Since components send the groups they are interested in request messages, only those that participate in the informed groups will respond. In fact, unicast discovery is also a particular kind of active discovery. On passive discovery, DAs regularly send announcement messages, also known as "I am alive" messages, representing its availability. In this case, UAs and SAs select among DAs instances announced, those representing some group of interest. Indeed, passive discovery enable components to test DA's availability in a similar mechanism as the negative acknowledgment in request-response protocols [?].

Although some SDP protocols like Jini and CORBA Trader enforce the use of DAs, in some protocols they are optional or even do not exist, for example UPnP (Universal Plug and Play) [9]. So this work envisions two trader configurations as shown on Figure 2: one using DA (2.a) and the other (2.b) without them. We figure out the peer-to-peer configuration as a refinement of the DA-based architecture where the remaining components assume tasks of the absent DA. On peer-to-peer configuration SAs frequently announce their availability by means of multicast announcement messages in a very similar approach as passive discovery. On the other hand, UAs select only those services of interest.

Peer-to-peer is simpler than DA-based trader and, in some contexts, is also considered more reliable since DAs introduce another failure point and also the opportunity to inconsistently represent service properties. Moreover, DA represents a single communication point which parties must contact prior to start direct communication with a target service, thus introducing some overhead. On the other hand, regarding to communication overhead, peer-to-peer traders span multicast messages proportionally to the number of network services. In DA-based traders, this number is proportional to the number of DAs, which is theoretically less than the number of services. In addition, a directory-based architecture allows direct reach-ability of the name services by the unicast discovery protocol, which has an important role on the current Internet where there not enough support to multicast communication.

Therefore, peer-to-peer traders are being considered well suited to small-scale networks with a few numbers of services, like home or automobile networks. UPnP and Bluetooth [6] are examples of peer-to-peer traders, the later being specific for short range wireless networks. On the other hand, repository-based (DA) traders seems better suited to large-scaled networks, like the Internet. [2] makes a comparison among Jini and CORBA traders, the two main representatives of repository traders where is posed that Jini presents better tradeoffs than CORBA in regard to Internet trading. The opportunity to download a full-fledged proxy object rather than a remote reference, and the absence of a centralized service type repository are some arguments that justify such conclusion. Main features of currently relevant SDP technologies are illustrated on Table 1.

The following subsection briefly introduces Jini's architecture and principles.

3.1 Jini Principles and Architecture

The Jini architecture is divided into three categories: *infrastructure*, *programming model* and *services*. Discovering communities, joining services on these communities and searching for services are covered by the infrastructure layer which is responsible to provide minimal conditions for services to get into Jini networks.

The programming model defines a set of API's that enables the construction of reliable services. While the first layer concerns with infrastructure issues such as service availability and location, this second layer covers application domain problems in a distributed context

Features	SLP	Jini	UPnP	CORBA
Institution	IETF	Sun	Microsoft	OMG
Net.Transport	TCP/IP	independent	TCP/IP	IIOP
Prg.Language	independent	Java	independent	independent
Code Mobility	no	yes	no	no
Attribute Search	yes	yes	no	yes
Leasing	yes	yes	no	no
Security	IP based	Java based	IP based	COS
DA	optional	yes	no	yes

Table 1: SDP features

such as fault-tolerance (Leasing Service), asynchronous communication (Events Service) and distributed consistency (Transaction Service). Despite all these mentioned features may also be implemented as services, the last component of this architecture holds the remaining services that are not in the infrastructure or the programming model set of services. In fact, some of them have already been standardized, such as the JavaSpaces service. The Jini architecure is illustrated in Figure 3.

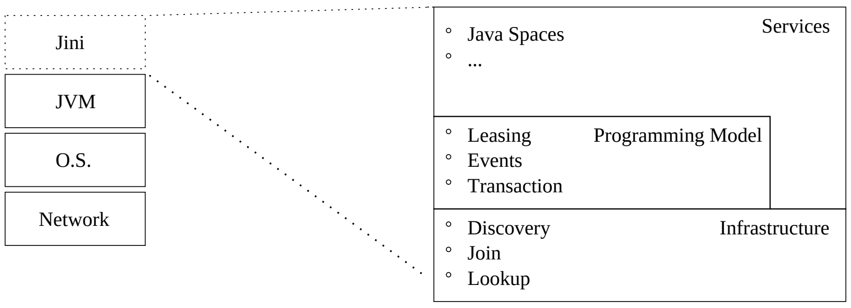

Figure 3: Jini architecure

Within a community, services interact with one another either as clients or servers. To support this interaction, they must get references to themselves, which is accomplished with support from a special service, the name service, known in Jini as the Lookup Service. This basic service defines the available services in a Jini community, providing operations for service search and registration. Before invoking these operations, however, a new service joining an existing community must get a reference to a Lookup Service in that community. This process is defined by the *Discovery* protocol [1]. In addition, the Lookup Service also provides an administrative interface and plays a key role in federation of Jini communities.

Services are represented in Jini Lookup Service by an identifier, a proxy, and attributes. The proxy concept has fundamental importance in Jini. Like stub objects produced regularly by RMI, this object implements the service interface and are downloaded as needed to the client's address space. In contrast, this object does not necessarily need to implement the RMI `Remote` interface, but only the service interface. For example, the proxy may be implemented to access a CORBA object backend, or it may use several different remote objects, or even it may be implemented locally since it is a full-fledged regular Java object. Thus in regard to Jini network transport independence described on Table 1, the proxy can implement any protocol it needs, even those mentioned in that table. However, in order to implement the discovery protocol and download a proxy, Jini's current implementation depends on TCP unicast, multicast UDP (optionally), and RMI.

As JTrader is based on Jini, hereafter DA and "Lookup Services", also known in Jini terminology simply as Lus, are used interchangeably.

4 JTrader Design

This section presents a design for a service trader federation over the Internet where new services are installed every day. Traders seem to be very appropriate to this dynamic environment as it is based on an asynchronous and very discoupled distributed programming model that allows clients to discover services with very few information about them. For instance, there is no implicit order for starting components: first the server, next, the clients, which would mean a difficult item of configuration, considering the Internet global scale. Furthermore, trading over large scale networks enables users and programs to use services wherever they are - there's no locality constraint concerning the service or the clients.

Current SDP technologies resolve the problem about service discovery when the scope of searching is somehow reduced, for example, in an enterprise network [8], a store's wireless network or an automobile network. As mentioned before, the enforcement to use multicast communication hinders the opportunity to a seamlessly migration to larger scale networks, like the Internet.

JTrader presents a Jini-based software solution that enables services to be registered transparently in an Internet service federation, almost without system administration efforts. In addition, users and programs, as represented in Figure 1, can access services as if they were on their own local networks in such a way that semantics of DA groups defined on those remote networks are preserved. Therefore, JTrader gathers public remote service proxies and registers them on its DAs, providing a means for localizing global services wherever they are hosted. In contrast to the peer-to-peer architecture, this approach somehow centralizes communication in order to make possible universal reachability, nevertheless it does not maintain any kind of service's state[3].

From now, the JTrader coarse-grained architecture and core components are presented:

- *JTrader Federation*-JTF. The federation manages sets of DAs what, in Jini terminology, is called Lookup Service or simply Lus. As a consequence, it defines a policy governing service proxies registration as a means to provide load balance and scalability. This is the core system component on which every service globally accessible is registered. Taking a simple design as a requirement, this component defines only two operations: `lookup` and `register`. To some extent, the former represents a discovery protocol version over the Internet, by which programs could inform a group and the component retrieves the set of DAs joining that group. At first glance, the `register` operation could seem unnecessary since DA already implements it. While describing federation details on next section, we justify the reason for this operation requirement.

- *JTrader Federation Adapter*-JTFA. Since the federation exposes public services over the Internet, distributed application can access them by means of a remote adapter object located in the client's address space. By definition, an adapter component

[3]Although service offer properties are also considered part of the service state, Jtrader keeps them consistent with the service remote implementation

implements an interface known to its clients without having to assume what class is used to implement that interface [5]. Figure 4 illustrates this design pattern using JTrader components.

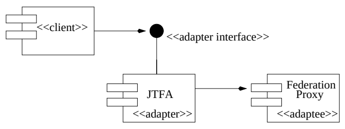

Figure 4: The adapter design pattern

Indeed, the adaptee instance is unknown until client bootstrap time when the adapter object looks up a federation object. The adapter component hides the programmer details of searching for the federation, once the federation is found, its proxy is bound to the JTFA component and method calls to this adapter component are delegated to the remote federation.

Discovering the JTF location is not a problem to the Web system described ahead, since, in the current configuration, JTF and JTWS live on the same network, so that the Web system can use multicast to dynamic discover the federation. In fact, this is a very specific case. In order to resolve the location problem to the Internet, the current adapter implementation statically defines a set of primary name servers where the federation proxy object can be retrieved. This can lead to failures when the primary services mentioned crash or change their location. Despite the adapter design pattern used allows changing this set without affecting client and server code, it requires manual reconfiguration, which means a very hard requirement. We are working on alternative solutions to handle this problem.

- *JTrader Web System*-JTWS. This component provides a Web front-end to users access terminal services available on the federation. Regarding to Figure 1, this components interfaces with the user actor. Using this component, it is possible to define templates as if you did it programmatically, and also submit queries, and receive results. These results represent service matching. Therefore, users may download service's programming interface and use the JTFA component to access some service. For those services which provide some supportable[4] user interface, users can also access and operate the service *on-the-fly* using the Web as a starting point without requiring to download programming interfaces and using the JTFA component outside the Web System. Therefore, Figure 5 introduce the software components that interface with the federation.

- *JTrader Enterprise Agent*-JTEA. This component represents a remote stationary agent installed on a network that hosts services to be published in the federation. The common use of this component is for enterprises that want to achieve scale on using their service components. Not every service installed on this network have to be published in the federation, but only those who provide some complete functionality to users and programs - when it is a subsystem. Once installed on the network, this agent listens for service state changes; thus if a service crashes, starts or modifies some property; the agent, by means of its remote adapter, should notify the federation in order to

[4]JTrader only provides support to URL-based UI like Java Servlets, Microsoft ASP, and JSP

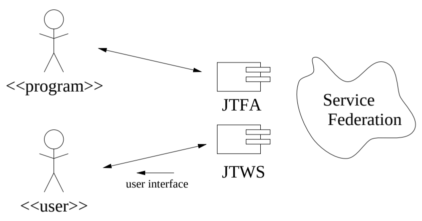

Figure 5: The federation's client components

keep service offers in consistent state. In fact, service crashes are handled in a different approach. As service registration is leased, failing to renew the lease make registration to be spontaneously dropped. This situation may happen on agent or service failure. In both cases, the federation is automatically recovered to a consistent state without requiring any communication. Figure 6 enlightens the organization of this components and their connections.

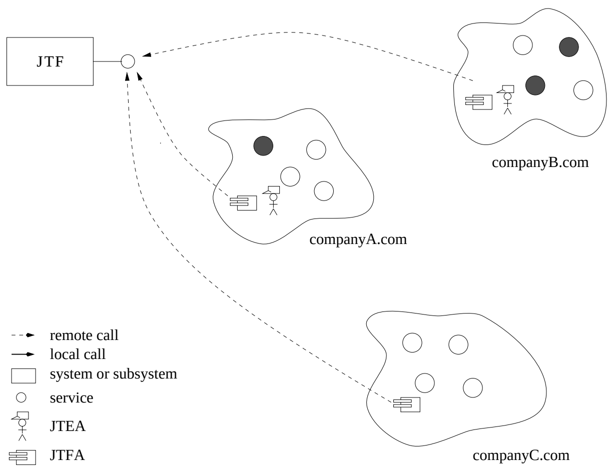

Figure 6: The remote adapter and enterprise agent

5 Federation Details

Designed for large scale use, the federation component - JTF - should meet scalability, availability and performance requirements. The system should keep running even in presence of partial failures, searching for alternative running instances of failed objects and dynamically binding them to the federation - *availability*. The federation should be able to grow, adding new service instances, name servers, and hosts, in order to support access and registration high volumes - *scalability*. Moreover, the federation has to achieve an acceptable overall efficiency by means of conscious balanced compromises - *performance*.

SDP protocols very often provide a means to filter, while in discovery phase, name servers which join on some well known group. Considering system administrators define suitable groups representing service categories some enterprise has, it is possible to discover specific DAs, whose registered services have some similar semantic pattern like "devices". In such a case, the result of discovering DAs joining on this group is that proxies enabled to access devices are registered on them, so query scope is confined. However, this feature is very hard to be achieved on a considerably distributed scenario whereby programs could really share a service marketplace. Imagine, for example, "companyA" uses "devices" to name the devices group, while "companyB" uses "Devices". Actually, this is a hard problem to be resolved both for clients and services. In such a case, clients do not know which string should be used to name the device group, and services are enforced to agree on a shared ontology in order to classify their objects on a single group. Despite we do not present or recommend any shared service ontology, this work assumes the existence of one.

Even with this assumption, the federation faces a hard problem about resource allocation. To confine services in a common place, the federation should have at least one running Lus per service category. This is infeasible regarding the potential number of enterprise groups and the intermittent frequency, which they are used on JTF. Therefore, concerning the federation scale, this behavior is very hard to achieve since the number of potential groups defined in remote networks (enterprise) is supposed to be some orders of magnitude greater than the number of Lookup Services hosted in the federation. On the other hand, allowing a Lookup Service to join on multiple and semantically distinct service categories deny the principle of segregating services to scope queries and administration. In fact, JTF does not conform with this principle strictly. It controls which Lus a service is to be registered by providing a front-end interface to register the service, and thereby the registration is performed transparently to services. Services from different groups may actually share the same Lookup Service. Therefore, that group used in local Jini networks to search for Lus instances containing only a specialized kind of services does not hold anymore, rather these groups are managed by the federation to know exactly which user-defined groups maps to which Lus instances and reversely.

On the other hand, as in regular Jini networks, the federation system strictly defines which groups the Lus join, but these groups do not represent service categories, rather they are groups assigned by the system administrator to enable load balancing and raise the system availability. Indeed, Lookup Services joining on the same, here called, physical group are configured to keep the same set of services regardless if they join on distinct user defined groups. Thus, JTrader uses two kinds of groups as described bellow:

- *user-defined group.* This group is dynamically managed by the JTF component to provide group awareness to JTrader's users. In contrast to the Jini approach of attaching static groups to Lus, this approach attaches services to groups as a consequence of using a Lookup Service to store offers from multiple user-defined groups. These groups are transparent to Lookup Services, only the federation users access them.

- *administrator defined group.* A physical and administrative group used to multicast discovery Lookup Service instances into JTrader network. Lus instances joining on the same physical group should keep the same set of services. The consistent replication of services is achieved by tunnels, as shown in [4], installed between Lookup Services joining the same physical group. These groups main purpose is to balance the system load and increase service availability, so they are transparent to users. Therefore, any combination of Lookup Services which has at least one component from each physical group represents the whole federation and the load balancing solution should guarantee the best selection.

As represented in Figure 7, this arrangement enables unprecedented interchange among Lookup Services as long as some instance fail or become somehow inefficient. A directed tunnel between a pair of Lookup Services, causes any modification in one Lus, like a new service being registered, be also replicated to the other. The only constraint is to avoid loop on the tunnel network so that messages would be never dropped. However, this problem can be resolved tagging the messages posted by the initial Lus with an identifier, if the message reaches the source, it will be dropped. Starting new DA instances, attaching these instances to groups, defining tunnels among them, and monitoring services registered are features of an independent administration tool called: JTrader Admin[5].

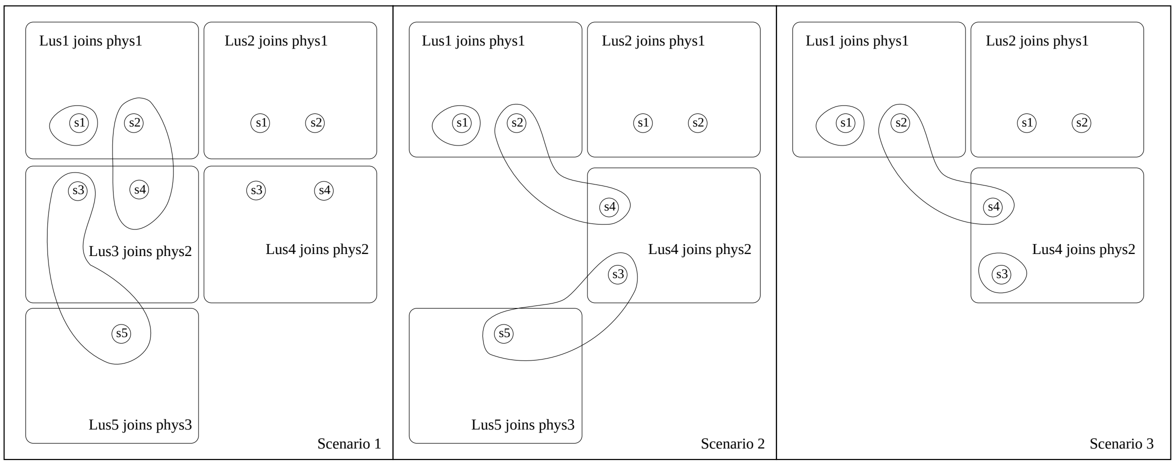

Figure 7: user-defined and physical groups

In Figure 7 three configuration scenarios are presented where five service offers are registered in three distinct user-defined groups. The initial configuration (Scenario 1) has five Lookup Services joining in three physical groups, and also proxy tunnels installed as mentioned earlier: Lus1 $\leftrightarrow$ Lus2, Lus3 $\leftrightarrow$ Lus4. The second scenario presents an automatic reconfiguration after a Lus failure whereby the replicated Lus start to respond for services registered in the failed one. The third scenario also presents a Lus failure situation, but in this case the Lookup Service had no replication. In fact, this situation may also occur in Jini enterprise networks and there is no trivial solution to this problem as well. Services can catch the event of a Lus failure and register its offer in a different group. However, this approach does not seem a reasonable alternative, as it will probably be very difficult to clients find that service in a different group. In other words, if a Jini networks does not provide at least one Lus associated with a given group, we naturally expect that no service joining that group could be registered, and service offers would be lost even if the service is available. Therefore, the system should guarantee there are at least two running Lus for the same physical group in order to avoid service offer lost, which is graphically represented in the transition between Scenario 2 and 3 by a user-defined group reduction. Also, this function could be well automated by the JTrader Admin.

In a Jini private network, a service can register to receive events on Lus failure and try, in response, to discover a new available one that joins the same group. In contrast, JTrader uses a different approach in order to provide service availability and reduce communication with the federation. Using tunnels, JTF makes a best effort to keep services proxies available. If a proxy is unregistered it is due to failing in renewing a lease or if there are no more Lus available on the physical group, as presented earlier. In other words, federation clients will not use a Jini `DiscoveryListener` instance to manage Lookup Service availability events.

[5]At this moment, this auxiliary tool is not implemented.

In order to increase reliability, JTrader system should provide redundant federation objects that would assume the place of the other in the event of a failure. However, this feature is not yet developed since it is very related to the presence of a persistence service, which this system does not currently use. This persistence service should be synchronized to allow federation services concurrently access the system's state. On the other hand, storing the state on a remote place should introduce some overhead if compared to the state persisted locally. The final chapter discusses this theme in more detail.

5.1 API

As mentioned earlier, the JTF component can be considered as a special case of discovery protocol as it provides a means to access DAs. The federation interface also defines a `register` operation which could seem unnecessary considering the Lookup Service interface - `ServiceRegistrar` - defines a similar operation. However, using the Lookup Service `register` method makes sense in environments that users have some influence over DA creation like enterprise networks; but this is not the case on wider networks which user-defined groups are created on a demand basis. When searching a Lookup Services of a given user defined group for the first time, probably, no instance will be returned by the federation, thereby service offers could not be registered if the `register` operation were not provided on JTF interface. In addition, this operation enables JTF to associate user-defined groups with Lookup Services. The JTF interface is defined as follows:

```
   public interface IJTrader {

      public ServiceRegistration register(String group,
                                          ServiceItem item,
                                          long leaseDuration)
          throws RemoteException;

      public ServiceRegistrar[] discover(String group)
          throws RemoteException;

}
```

Once services are successfully registered on JTF, Lookup Service proxies may be retrieved by calling the `discover` method. Therefore, as Lookup Services are localized, client components do not need to interact directly with the federation object anymore, thus alleviating JTF from concurrent access. Notice that more than one Lus proxy may be returned in a `discover`() call as user-defined groups may use more than one Lookup Services. This happens when the system is regarded as overloaded by applying a heuristics to verify this situation. The load balancer component is discussed on the next section. In fact, this decision may affect the client behavior as the Lus instances returned represent disjoint sets of services, and thereby a client should query each Lus untill find the intended service. However, we will see the system can be configured to not use this feature, thus the `discover`() operation would return only one Lus instance and services joining on some user-defined group will not span over multiple Lus.

5.2 Load Balance

By starting replicated Lus, the federation increase reliability since lost of services is avoided when Lus instances fail. However if a Lookup Service becomes overloaded due to a high volume of service offers, the replicated intance should be as well overloaded. Thus, physical groups do not provide a natural mechanism to balance the load.

The JTF component is not directly responsible to implement load balancing, rather it delegates these functions to the `Balancer` subcomponent. The `Balancer` is extensively used by the federation to select the Lus that will be used to register a service, meaning that services from the same user-defined group may be spread in different physical groups, as represented on 7. In fact, threre is a tradeoff between keeping user-defined group in a small set of physical groups or larger ones. Enlarging this set may increase load balance in detriment of performance and transparency caused by the distribution of service offers - remember clients should fetch services from disjoint Lookup Service instances.

As a requirement to load balance, services from the same user-defined group should be confined in a restricted and configurable upper bound number of physical-groups. When this number is assigned to `1`, the balancer should not attempt to balance the load of the system, as a consequence, user-defined groups do not form disjoint sets and `discover` operation returns only one Lookup Service instance that should be any of those representing the same physical group initially used. JTF always asks the `Balancer` for the *best current* Lus, however, the `Balancer` does not always answer with the best one as it must apply some rules in order to control resource misuse. For example, if Lus1 already has a service from some user-defined group, the next service from that group is likely to be registered also on Lus1 if this Lus1 is not regarded as overloaded. Currently, load evaluation is only based on Lookup Service size - the number of offers registered. Moreover, even though the heuristic the `Balancer` uses to accomplish this selection is briefly presented bellow, this work does not intend to present a definitive solution on this theme, so this component is properly decoupled from the JTF, but only a general model to manage a dynamic federation. By the way, next chapter poses that Lookup Service size is not a good parameter to evaluate the load of the system.

- The evaluation difference between the best Lus joining on some user-defined group and the *best current* one should be greater than some configurable value. In other words, this rule says that a new Lus should only join on a user-defined group when it is significant better than the older.

- The number of Lus already joined on an user defined group should be less than some maximum *absolute* value and also less than some maximum *relative* value, both configurable. The first number imposes an absolute upper-bound to user-defined group size, however it does not scales well when the number of Lus in the federation is considerably small. Thus, a number relative to the number of running Lus should be also defined.

- If the above conditions are not met, JTF uses the best Lus already joining on the user-defined group

The organization of JTF components is shown in Figure 8:

5.3 Current Shortcomings

Trying to scaling the trading service to the Internet, this work faced two hard problems: federation dynamic location and security constraints imposed by the network. As long as

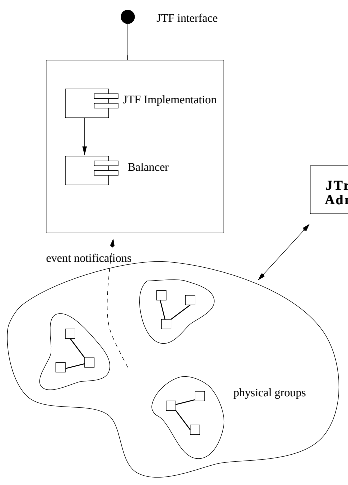

Figure 8: The JTrader federation

multicast communication is a very hard issue in current Internet, this work provides a solution based on unicast. Actually, multicast communication is very limited concerning IPv4 and this work is based on current Internet. The JTFA component has used a list of alternative primary name servers by which the federation object is likely to be registered. Using name servers causes a dependency between the client and the federation, say the federation adapter and the federation, as it should be localized by some specific identifier - the name of the server, for instance. Therefore, the federation location is not completelly transparent to the adapter component as the location of the name server is not. Furthermore, the RMI specification enforces the service to be running on the same location as the name server, so the federation object is supposed to be running on a host of the mentioned list.

The application multicast tunneling approach delegates to a process that listens for unicast messages on the Internet and then forwards multicast messages on the local network. As the federation object is installed on a Jini network, the tunnel process is able to reach its proxy by using the Jini multicast discovery protocol. Therefore, the federation object is allowed to be running in another host than the name server host. However, the adapter component still needs to statically discover the tunnel process by reaching the name server where it is registered.

In addition, the Internet has a multitude of network policies and constraints regarding to the hostile environment where it lies on. Concerning this scenario[6], security is another hard issue for distributed computing as it must be enabled without opening ports for hackers. Also, it is very difficult to find subnets with similar security policies, what makes general solutions very hard facing the network scale. This way, achieving scale and handling security are contradictory issues on Internet distributed systems. In [3] we has detailed the security solution approached by the JTrader system.

6 Conclusion

This work has introduced current SDP technologies, presented some scenarios of use, and finally has described the design of JTrader, an approach to service trading federation over

[6]In fact, a great deal of users access Internet by means of ISP through cable modem and POP3 connections

the Internet. The JTrader system is made of four components: *JTrader Federation* is the core component that manages a set of Lookup Services where currently available services are registered, this component aimed at providing a highly scalable and reliable service from where worlwide service offers can be retrieved; the *JTrader Federation Adapter* provides a means to programs transparently access the federation on remote locations; the *JTrader Web System* is a Web front-end that provides tools to search for services on that federation and perhaps using them; and finally *JTrader Enterprise Agent* provides a means to enterprises publish their services almost without administration.

References

[1] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, December 1999.

[2] Marcelo d'Amorim, Vander Alves, and Carlos Ferraz. A Comparative Analysis of Trading approaches. Universidade Federal de Pernambuco, November 2000.

[3] Marcelo d'Amorim and Carlos Ferraz. Leveraging Security in Internet Distributed Applications. Universidade Federal de Pernambuco, February 2001.

[4] W. Keith Edwards. *Core Jini*. Sun Microsystems Press, June 1999.

[5] M. Grand. *Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. John Wiley & Sons, New York, NY, USA, 1998.

[6] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen. Bluetooth: Vision, goals, and architecture. *ACM Mobile Computing and Communications Review*, 2(4):38–45, Oct 1998.

[7] James K. Watson Jr., Jeetu Patel, and Joe Feener. *Free Trade Zones*. Doculabs, July 1999.

[8] James Kempf and Pete St. Pierre. *Service Location Protocols for Enterprise Networks*. Wiley, 1999.

[9] Microsoft Corporation. *Universal Plug and Play Device Architecture Reference Specification*, November 1999. available at http://www.microsoft.com/hwdev/UPnP.

[10] Object Management Group. *CORBA OMA Specification*, June 1986.

[11] Sun Microsystems. *JavaSpaces Service Specification*, 1.1 edition, October 2000.

[12] V. Vasudevan. *A Reference Model for Trader-Based Distributed Systems Architectures*. Object Services and Consulting Inc., January 1998.

[13] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. Tspaces. *IBM Systems Journal*, 1999.