Carleton University, Systems and Computer Engineering, Technical Report SCE-03-16, July 2003

A Survey of Middleware Paradigms for Mobile Computing

Abdulbaset Gaddah and Thomas Kunz

Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive
Ottawa, Ontario, Canada K1S 5B6

{agaddah, tkunz}@sce.carleton.ca

Abstract

Current advances in portable devices, wireless technologies, and distributed systems have created a mobile computing environment that is characterized by a large scale of dynamism. Diversities in network connectivity, platform capability, and resource availability can significantly affect the application performance. Traditional middleware systems, like CORBA and DCOM, have achieved great success in dealing with the heterogeneity in the underlying hardware and software platforms, offering portability, and facilitating development of distributed applications. However, they are not prepared to offer proper support for addressing the dynamic aspects of mobile systems. Modern distributed applications need a middleware that is capable of adapting to environment changes and that supports the required level of quality of service.

This paper represents the experience of several research projects related to next generation middleware systems. We first define middleware and indicate the major challenges in mobile computing systems. We then take a broader perspective and try to identify the main requirements for mobile middleware systems. Following this, we review the different categories of mobile middleware technologies and show their strength and weakness. We finally present a simples discussion on the surveyed work and provide a number of observations about the remaining issues.

1 Introduction

The availability of lightweight, portable computers and wireless technologies has created a new class of applications called *mobile applications*. These applications often run on scarce resource platforms such as personal digital assistants, notebooks, and mobile phones, each of which have limited CPU power, memory, and battery life. They are usually connected to wireless links, which are characterized by lower bandwidths, higher error rates, and more frequent disconnections.

Most distributed applications and services were designed with the assumption that the terminals were powerful, stationary and connected to fixed networks. Conventional middleware technologies thus have focused on masking out the problems of heterogeneity and distribution [44] to facilitate the development of distributed systems. They allow the application developers to focus on application functionality rather than on dealing explicitly with distribution issues. Different middleware systems such as CORBA, DCOM and Java RMI have proved their suitability for standard client-server applications.

However, under the highly variable computing environment conditions that characterize mobile platforms, it is believed that existing traditional middleware systems are not capable of providing adequate support for the mobile wireless computing environment. There is a great demand for designing modern middleware systems that can support new requirements imposed by mobility.

This survey provides a general overview of the most relevant mobile middleware systems and highlights not only modern solutions but also goals that still need to be achieved. The main aim of the survey is to assist middleware researchers to evaluate the strength and weakness of proposed approaches and to gain some knowledge about the requirements of the mobile middleware systems.

This paper is organized as follows: in Section 2, we define the term *middleware* and it's primarily role in distributed systems. We also present the major challenges in mobile computing environments. Then, we identify a number of requirements for building next generation middleware. Section 3 illustrates a simple case study that highlights the necessity of adopting these requirements. Section 4 gives a detailed review of the major solutions proposed to date. One approach is the use of reflective middleware systems to support re-configurability and adaptability. This solution is based on the concept of open implementation and reflection to provide an access to the underlying system. We have reviewed a number of such middleware systems, including OpenCorba [1], Open-ORB [4], DynamicTAO [8], FlexiNet [11], and Globe [13]. Tuple Space middleware provides another solution that handles the problem of disconnected operations by supporting the asynchronous communication paradigm. Tuples are the basic elements of this category of middleware that can be read or written by many participants. We discuss some Tuple Space solutions, like LIME [14], TSpaces [16], and JavaSpaces [17]. Context-aware middleware exploits the concept of awareness to expose the internal and external execution context to applications that may direct the middleware to better performance. We present one of these middleware systems called Nexus [18]. We then review event-based middleware that provides the basis for building lightweight middleware with asynchronous interaction model. The Publish and Subscribe paradigm forms the general idea of this type of middleware that has been expressed in Hermes [29]. We also describe some other middleware solutions, Bayou [38] and Jini [41], which have been particularly developed to target specific mobility issues such as disconnected operations and service discovery. Following this, Section 5 provides a simple discussion on the surveyed work and presents a number of remaining issues. Finally, Section 6 draws our conclusions on the need of new approaches for next-generation middleware.

2 Mobile Architectural Requirements

Many people come up with different definitions of the term *middleware*; however, all agree

that it plays a vital role in hiding the complexity of distributed applications. These applications typically operate in an environment that may include heterogeneous computer architectures, operating systems, network protocols, and databases. It is unpleasant for an application developer to deal with such heterogeneous "plumbing". Middleware's primary role is to conceal this complexity from developers by deploying an isolated layer of APIs. This layer bridges the gap between application program and platform dependency. Middleware is defined as follows by Linthicum [45].

"Middleware is an enabling layer of software that resides between the application program and the networked layer of heterogeneous platforms and protocols. It decouples applications from any dependencies on the plumbing layer that consists of heterogeneous operating systems, hardware platforms and communication protocols".

2.1 The Limitations of Mobile Computing

There are at least three common factors that affect the design of the middleware infrastructure required for mobile computing: mobile devices, network connection, and mobility. ***Mobile devices*** vary from one to another in term of resource availability. Devices like laptops can offer fast CPUs and large amount of RAM and disk space while others like pocket PCs and phones usually have scarce resources. It is either impossible or too expensive to augment the resource availability. Hence, middleware should be designed to achieve optimal resource utilization. ***Network connection*** in mobile scenarios is characterized by limited bandwidth, high error rate, higher cost, and frequent disconnections due to power limitations, available spectrum, and mobility. Many wireless and mobile networks such as WaveLAN are organized into geographically defined cells, with a control point called a base station in each of the cells. Hosts with in the same cell share the network bandwidth; hence, the bandwidth rapidly decreases whenever a new host joins the cell. Mobile devices may move around different areas with no coverage or high interference that cause a sudden drop in network bandwidth or a loss of connection entirely. Unpredictable disconnection is also a common issue that frequently occurs due to the handoff process or shadowed areas. Most wireless network services charge a flat fee for their service, which usually covers a fixed number of messages. Additional charges are levied on per packet or per message basis. In contrast, the cost for sending data over cellular is based on connection time instead. This forces mobile users to connect for short period of time. ***Physical host mobility*** can greatly affect network connection, which accordingly has to adapt to user mobility by reconnecting the user with respect to a new location. Mobile clients may interact with different types of networks, services, and security policies as they move from one area to another. This requires applications to behave differently to cope with dynamic changes of the environment parameters.

Due to these limitations, conventional middleware technologies designed for fixed distributed systems are not prepared to support mobile systems. They target a static execution platform where the host location is fixed, the network bandwidth does not fluctuate, and services are well defined. We next identify a number of important requirements that must be provided by middleware for mobile computing.

2.2 Analyzing the Requirements for Mobile Computing

During the system lifetime, the application behaviour may need to be altered due to dynamic changes in infrastructure facilities, such as the availability of particular services. ***Dynamic***

reconfiguration is thus required and can be achieved by adding a new behaviour or changing an existing one at system runtime. Dynamic changes in system behaviour and operating context at runtime can trigger re-evaluation and reallocation of resources. Middleware supporting dynamic reconfiguration needs to detect changes in available resources and either reallocate resources, or notify the application to adapt to the changes. ***Adaptivity*** is also part of the new requirements that allows applications to run efficiently and predictably under a broader range of conditions. Through adaptation a system can adapt its behaviour instead of providing a uniform interface in all situations. The middleware needs to monitor the resource supply/demand, compute adaptation decisions, and notify applications about changes. ***Asynchronous interaction*** tackles the problems of high latency and disconnected operations that can arise with other interaction models. A client using asynchronous communication primitives issues a request and continues operating and then collects the result at any appropriate time. The client and server components do not need to be running concurrently to communicate with each other. A client may issue a request for a service, disconnect from the network, and collect the result later on. This type of interaction style reduces the network bandwidth consumption, achieves decoupling of client and server, and elevates system scalability. ***Context-awareness*** is an important requirement to build an effective and efficient adaptive system. The context of a mobile unit is usually determined by its current location which, in turn, defines the environment where the computation associated with the unit is performed. The context may include device characteristics, user's activities, services, as well as other resources of the system. Context-awareness is used by several systems; however, few systems sense execution context other than location. The system performance can be increased when execution context is disclosed to the upper layer that assists middleware in making the right decision. ***Lightweight middleware*** needs to be considered when constructing middleware for mobile computing. Current middleware platforms like CORBA are too heavy to run on devices with limited resources. By default, they contain a wide range of optional features and all possible functionalities, many of which will be unused by most applications. For example, invoking a method on a remote object involves only client side functionality and either Dynamic or Static Invocation Interface. Most of the existing ORB implementations provide either a single or two separate libraries for the client and server sides that contains all functionality. This forces the client program to be glued with the entire functionality without having a choice to select a specific subset of this functionality. To get better insight to all the requirements mentioned above, we illustrate a simple case study in the next section.

3 The Motivations for the New Requirements: a Case Study

This section presents an electronic home shopping/bidding application as a case study that motivates the need for new middleware solutions that support the various requirements discussed previously. In this example, we particularly aim to highlight the different degrees of complexity introduced by mobility and the role of the new requirements for developing better middleware systems.

As shown in Figure 1, customers in the bidding system can access information about different items found on various online e-commerce sites using PCs or Pocket PCs (PPCs). The prices of products hosted on the e-commerce sites can either be fixed or left subject to competitive bidding in a real-time auction. We consider a scenario where a particular customer uses the PC to access the sites at home and the PPC when he moves around. We assume the PCs are

usually resource-rich and have high Internet connectivity and bandwidth. This kind of environment allows customers to access data from the sites when needed with low computational and network latencies. In contrast, mobile devices such as Pocket PCs are resource-poor and are usually connected to the Internet through poor quality links. This makes frequent access to the information on the sites unaffordable. Data replication is a common solution suggested for such a problem. In our example, the products' information needs to be replicated locally and updated whenever the availability or the price of items is modified. Due to memory/disk limitation, PPCs cannot support replicating the entire information locally. Thus, the products' information can be replicated on nearby PCs and accessed either from these PCs or directly from the e-commerce sites. With current middleware the replication is done transparently to the client application. Such transparency hides the execution context (e.g., available disk or memory space) from client applications, which only know about the data they need to access, and hence cannot influence the replication decision. Location awareness is another form of context awareness that client applications need to support. It may happen that a customer wants to collect his order on his way home. Then, the application needs to be aware of its location, locate the nearby store, and finally send the order.

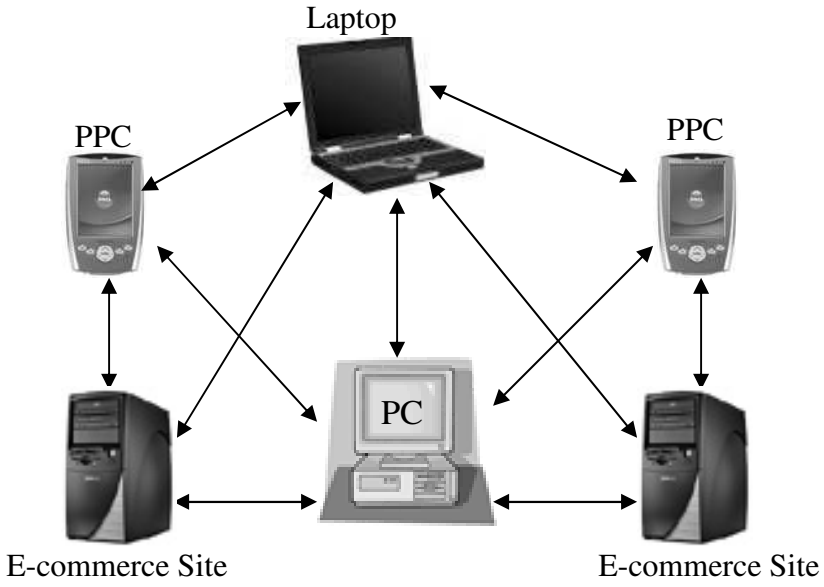

Figure 1: The Electronic Home Shopping/Bidding System

To get an impression of the dynamic reconfiguration functionality, we assume that a customer uses his mobile device to order two different products that are available at two different servers. We also assume that each server runs a different middleware platform (e.g., CORBA and RMI) and they each run a single instance of an object that provides their services. Therefore, the mobile device needs dynamic configuration that allows it to support method invocation mechanisms for RMI and CORBA objects. The middleware running on the mobile device needs to discover the two platforms, retrieve their properties, and reconfigure itself to generate an appropriate request.

Adaptivity is another key requirement that allows applications to react efficiently to various changes of conditions. In our example, we have focused on network bandwidth as one of

many candidates for adaptivity. Let us assume that a customer submits a query to a particular site. It may happen that the site's server is overloaded and hence is not very responsive. If we assume that the data is replicated on another server, the application may adapt to this situation by switching to the other server and thereby preserve the consumption of different resources (e.g., bandwidth, CPU, and battery). Now we assume that the user indicates in his request that he is interested in receiving a picture or a video clip of the product. The application can adapt to this case by activating a filtration protocol and passing the incoming data through it. The filtration process may include compressing, resizing, and dropping non-essential frames to reduce the bandwidth requirements.

The most common interaction model for existing middleware platforms is synchronous method invocation: a caller remains passive until a receiver responds. This is inadequate for mobile systems in which high latency and disconnected operations occur more often. As an example, we consider the scenario where a seller adds a new item to a selling list or a customer puts a bid on an item. This will involve the registration of the item on a site, notifying the interested customers, and finally returning a notification receipt. The asynchronous model is more convenient to use since the seller and customer will not be blocked until they receive the acknowledgement. They may continue doing other work and collect the notification receipt later. Also whenever a customer inquires about a certain product, a search process takes place to match the product on a site. If a synchronous model is all that is available, customers will perceive some delay due to the matching process. The delay may increase if it happens that the server holding the database is busy. Moreover, the matching process needs to be repeated again if disconnection occurs before receiving a response.

This small case study motivated a number of mobile middleware requirements. The next section illustrates several proposed middleware technologies and highlights their strengths and limitations with respect to these requirements.

4 Mobile Middleware Technologies

This section sheds some light on the different types of mobile middleware technologies, including their architectures, characteristics, and limitations. We start by introducing a classification that allows us to contrast and evaluate the different categories. Among the middleware systems we reviewed, we have identified four categories of middleware. Each category aims to support at least one of the above requirements imposed by mobility. These categories are reflective middleware, tuple space, context-aware middleware, and event-based middleware, each of which attempts to address the previous requirements using different approaches. Other middleware solutions have been reviewed and discussed in a separate category. They have targeted specific problems such as disconnected operations and service discovery. The following table illustrates how various requirements are met by the different categories.

Requirements	Reflective	Tuple Space	Context-Aware	Event-Based
Synchronous/ connection based	X		X	
Asynchronous/ connectionless		X		X
Re-configuration	X			
Adaptation	X		X	
Awareness	X		X	
Light weight				X

4.1 Reflective Middleware

The reflection technique was initially used in the field of programming languages to support the design of more open and extensible languages [19]. Reflection is also applied in other fields including operating systems [20] and more recently distributed systems [21]. The principle of reflection enables a program to access, reason about and change its own behaviour. Smith [22] defined the concept of reflection in the following quote:

"*In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures*".

A reflective system consists of two levels referred to as meta-level and base-level. The former performs computation on the objects residing in the lower levels. The latter performs computation on the application domain entities. The reflection approach supports the inspection and adaptation of the underlying implementation (the base-level) at run time. A reflective system provides a meta-object protocol (meta-interface) to define the services available at the meta-level. The meta-level can be accessed via a concept of reification. Reification means exposing some hidden aspect of the internal representation and hence they can be accessed by the application (the base-level). The implementation openness offers a straightforward mechanism to insert some behaviour to monitor or alter the internal behaviour of the platform. This enables the application to be in charge of inspecting and adapting the middleware behaviour based on its own needs. Thus, a lightweight middleware with a minimal set of functionality is achieved to run on mobile systems.

The main motivation of this approach is to make the middleware more adaptable to its environment and better able to cope with changes. Examples of middleware systems that adopted the concept of reflection are OpenCorba [1], Open-ORB [4], DynamicTAO [8], FlexiNet [11], and Globe [13]. We now review the major characteristics of some of these reflective systems.

4.1.1 OpenCorba

OpenCorba [1] is a reflective CORBA broker that provides dynamic adaptability of the ORB run-time behaviours (e.g. remote invocation, IDL type checking, and interface repository error handling). The OpenCorba architecture was implemented in the Smalltalk implementation NeoClasstalk [2] that supports metaclass programming for providing a clear separation

between what an object does (the base level) from how it does it (its meta level). The base level consists of classes that denote behaviour for their instances whereas the meta level consists of metaclasses that define behaviour for the classes themselves concerning object creation, encapsulation, inheritance rules, and message handling. To achieve dynamic modification, NeoClasstalk provides a protocol that enables objects to alter their class during run-time. This protocol is associated with the objects class to allow inspection of object behaviour during their life cycle and dynamic alteration of metaclasses to change a class property, at run-time. This can offer a "plug and play" environment to add and remove different class properties without regenerating code. Figure 2 illustrates an example of achieving dynamic adaptability in OpenCorba.

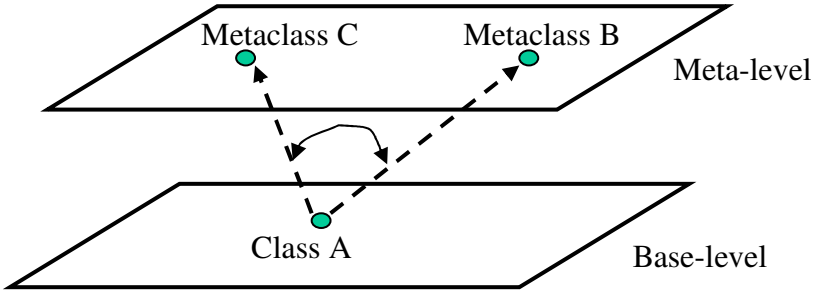

Figure 2: Dynamic adaptability of class properties

Originally, the metaclass B is associated with class A and then the association between A and B is changed towards metaclass C for data type checking as an instance. Eventually, class A retrieves its original state by altering its class association toward metaclass B.

The major reflective aspect of OpenCorba is the dynamic modification of the remote invocation mechanism through a proxy class. The IDL compiler of OpenCorba creates the proxy class (similar to client stub) and links it with the metaclass that intercepts the request and launches a remote invocation to the real server object. The invocation semantics can be easily redefined by associating the proxy with a different metaclass at run-time. This scheme is capable of introducing new mechanisms such as object migration or replication to improve the performances of distributed systems.

The performance issue in OpenCorba is mainly related to the performance of the message delivery, which depends on a technique called method wrappers [3] in NeoClasstalk. Method wrapping can be done at compile-time and run-time. At compile-time, it is difficult for the IDL compiler to anticipate which methods should be wrapped. The wrapper technique therefore needs to be applied to all methods, which involves extra cost. At run-time, OpenCorba introduces additional method invocations due to the metaclass indirection. The metaclass executes the invocation, which is received from the wrapper (this introduces extra invocation), to connect to the actual object server. This degrades the system performance since the indirection level is applied to every message send.

4.1.2 Open-ORB

Open-ORB [4] is a reflective middleware that has been implemented using Python. Open-ORB was designed to target configurable and dynamically reconfigurable platforms for applications that require dynamic requirements support (e.g. multimedia, real-time and

mobility). The principle of component, the unit of composition and independent deployment [5], is used in constructing Open-ORB middleware. The component model is derived from RM-ODP's computational model [6] where components have multiple interfaces for implementing various entities of middleware functionality. Each component associates with a meta-space. This meta-space is partitioned into a set of different meta-space models where each model deals with a different view of the middleware implementation. Open-ORB currently supports four meta-space models (the interface, architecture, interception, and resources meta-models) as illustrated in Figure 3.

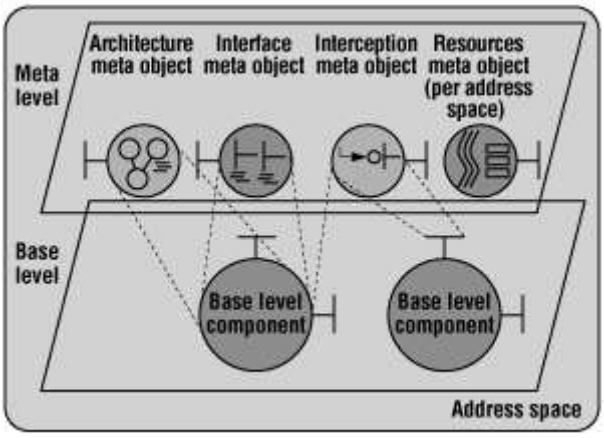

Figure3: The structure of meta-space in Open-ORB [4]

Each meta-model is accessed through an associated meta-object protocol (MOP) [7] that defines the services provided by the meta-model it represent. A component may associate with four meta-objects that provide access to the four meta-models. The structural reflection aspect is supported by the interface and architecture meta-models whereas the behavioural reflection aspect is supported by the interception and resources meta-models. The interface meta-model deals with the external representation of a component. Through interface definitions, the associated MOP can dynamically discover the services that are provided by a particular component. The architecture meta-model in turn deals with the internal representation of a component. The composition aspect is represented in form of a component graph where the interconnections between the constituent components are achieved by local bindings. The associated MOP provides operations to inspect and adapt the graph structure (e.g., add, delete or replace components), which offers dynamic adaptation. The interception meta-model is in charge of the execution environment for every component interface. It provides the ability of inserting additional functions (e.g., message arrival, enqueueing, dispatching, unmarshalling, scheduling, and thread creation) to an interface. The associated MOP manipulates these functions in form of dynamic plugable interceptors that perform pre- and post-processing of the interactions. The resources meta-model deals with the resource awareness and resource management of components that represent the underlying platform's resources along with their associated managers. The associated MOP enables the inspection and reconfiguration of the activities associated with the resources by adding/removing resources or altering the parameters/algorithms of the resource management.

The Open-ORB middleware architecture offers a promising step toward a robust design for next generation middleware platform and overcomes some CORBA limitations for supporting

multimedia. However, the middleware performance remains the major issue of the whole system. It is unwise to run Open-ORB middleware on mobile devices since it is built in top of a CORBA implementation that imposes a heavy computational load. Investigations on how to enable Open-ORB middleware to run efficiently on minimal devices, such as the Palm Pilot are still a matter for ongoing work.

4.1.3 DynamicTAO

DynamicTAO [8] is a reflective CORBA ORB, which extends TAO [9] to support run-time reconfiguration of the ORB engine and non-CORBA applications. It exports a meta-interface for transferring components across the distributed system, loading and unloading modules into the system run-time and inspecting and changing the ORB configuration state. In dynamicTAO, the internal system representation is exposed through a set of objects known as component configurators [10]. A component configurator stores the dependencies between ORB components and between ORB and application components. *DomainConfigurator* is an instance of a component configurator obtained by a process. It maintains references to instances of the ORB and servants that run within the process domain. *TAOConfigurator* holds hooks that are linked to different strategies such as concurrency, scheduling and monitoring. These strategies are realised as dynamically loadable libraries that can be attached to the running process. Figure 4 shows the main structure of dynamicTAO.

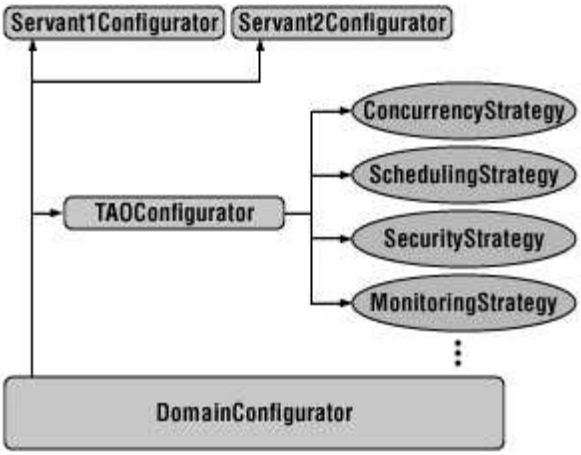

Figure 4: DynamicTAO structure [8]

Figure 5 illustrates the dynamicTAO architecture framework. Component implementations are stored in the local file system. The *persistent repository* is responsible for storing and manipulating the component implementations (e.g., browsing, deleting and creating). At run-time, a component implementation can be easily loaded from the local file system and linked to the associated process. A *network broker* triggers the reconfiguration action after receiving a request from a mobile agent that has been injected in the network. The mobile agent travels from one ORB to another, inspects and when necessary issues a reconfiguration request. The *network broker* then redirects the request to the *dynamic service configurator* who holds the *DomainConfigurator.* The *dynamic service configurator* provides a collection of operations that enable dynamic reconfiguration. Some of the functionality is assigned to *TAOConfigurator* or a particular *ServantConfigurator*. Other configuration tasks such as

ACE_Service_Config and *ACE_Service_Repository* are used to manage the start up configuration files and loaded implementations respectively.

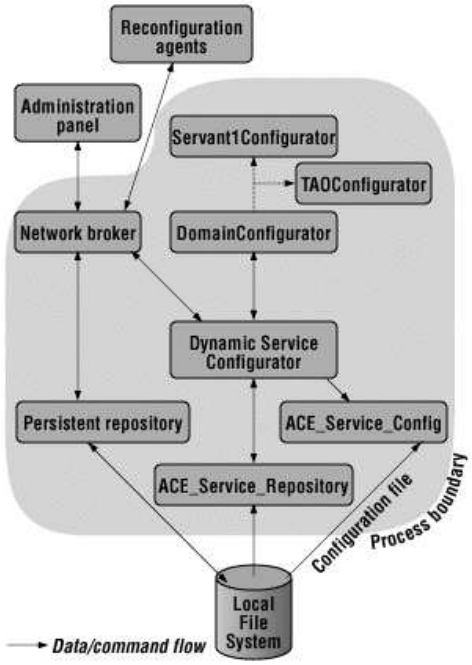

Figure 5: DynamicTAO components [8]

Supporting run-time reconfiguration raises two major consistency issues in dynamicTAO. Let us at first consider the scenario that involves replacing an old strategy $\mathbf{S}_{old}$ with a new one $\mathbf{S}_{new}$. The whole system could crash if unloading a strategy $\mathbf{S}_{old}$ occurred while it was being run by someone or a strategy $\mathbf{S}_{old}$ was invoked after it had been replaced. Thus, it is necessary to make sure that a strategy $\mathbf{S}_{old}$ is not in running state and it will not be called in the future. The second issue occurs when state information of a strategy $\mathbf{S}_{old}$ is required to be passed to a strategy $\mathbf{S}_{new}$ during the replacement. Dynamically loading and unloading components introduces another drawback in the system. Certain tools for achieving this mechanism are required. This subsequently increases the core's component size, which makes it unsuitable for mobile devices. By default most CORBA ORBs contain all possible functionality in a single library. This forces any application to be attached with the entire functionality even if the application only uses a subset of the entire functionality. Some CORBA ORBs split the client and server functionality in different libraries, but leave no choice for applications to select a particular subset of this functionality. Also, a substantial overhead is introduced by using Dynamic Invocation Interface (DII) whenever a client invokes a certain method.

4.1.4 FlexiNet

The FlexiNet platform [11] is a Java based middleware that places much emphasis on reflection and introspection at all layers. The layers of the FlexiNet protocol stack can be viewed as reflective objects that support different transformation on a method invocation. FlexiNet offers a generic binding framework and a set of meta-objects to implement basic RPC operations. FlexiNet stubs transform a simple invocation into a generic invocation and pass it through the protocol stack layers. Most of the stub functionality is shifted from stubs to

the protocol stack. This may simplify stub generation at run-time but increases protocol stack complexity. Different types of high level functions (i.e., serialisation, replication, object management) can be part of the protocol stack layers. The FlexiNet layers are designed to be highly autonomous to allow replacing or inserting extra layers. FlexiNet currently realizes only one protocol stack that is based on REX [12] (a reliable RPC protocol based on UDP). This protocol stack is generated by a single binder object called "green". Using Figure 6, we describe the different protocol layers, and follow the flow of a client request and a server response

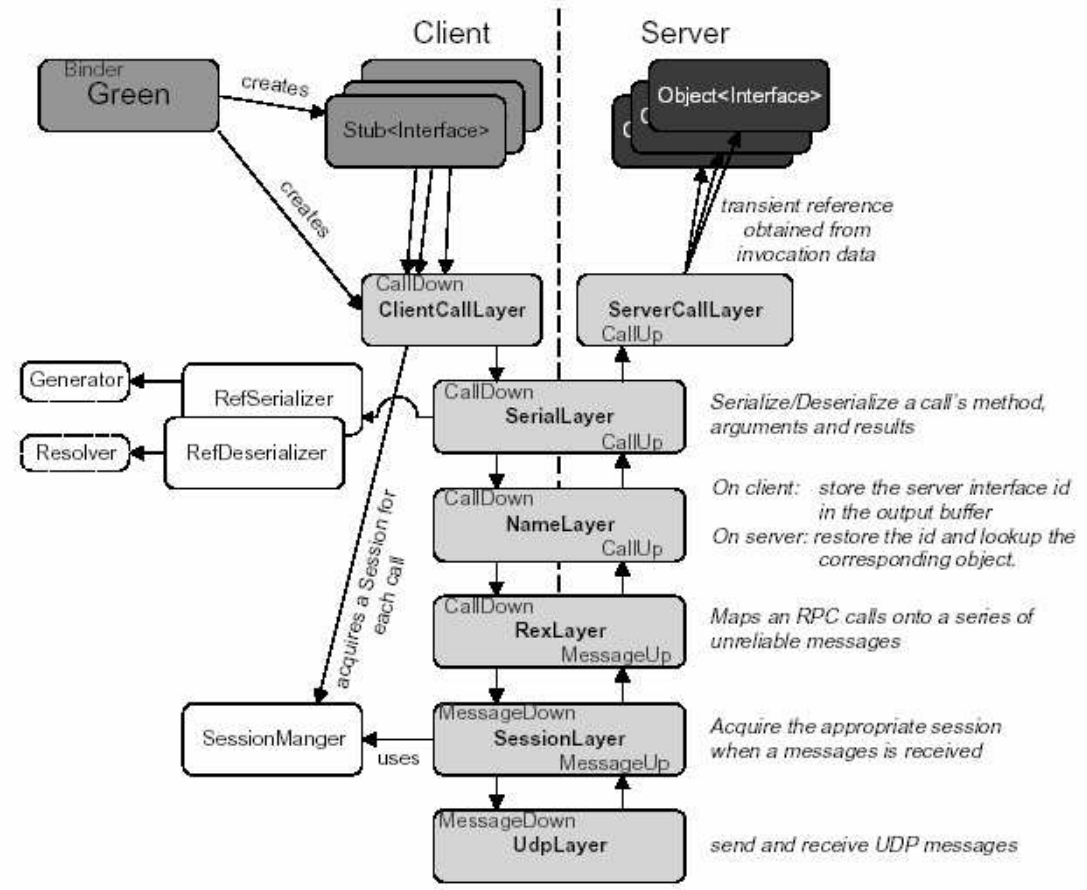

Figure 6: The protocol stack in FlexiNet [11]

The *CallDown* and *CallUp* interfaces are implemented by client and server layers respectively. A generic invocation can be passed to the next layer by calling *CallDown* or *CallUp* methods. Each layer processes the generic call if needed and then pass it to the next layer. This process continues until the call either reaches the bottom layer in a message form or the top layer in a call form. Two interfaces, *MessageUP* and *MessageDown*, need to be implemented in the RPC layer and downward layers. These two methods deal only with messages since the communication, below the RPC layer, is based on message passing. A call is initiated on the client stub, which converts the call to a generic representation. At this stage, argument validation and modification can be achieved and reflective classes may also be called. The "green" protocol currently does not provide reflective meaning; hence, the call is forwarded to the *ClientCallLayer* directly. This layer first contacts the *SessionManger* to

acquire a session object and then associates this session with call. The *SerialLayer* is responsible for serializing and deserializing methods along with their arguments and results. The *NameLayer* on the client side stores the server name identifier in the buffer. At the server side, it restores the server identifier to find the target object. The *ServerCallLayer* translates the generic call into a call on the correct target object. The *RexLayer* is used to map RPC semantics onto a series of unreliable messages. When the outgoing message reaches the *SessionLayer,* the session ID is encoded within the message. This will simplify re-establishing the session at the server side.

The last layer is a media access layer where sending and receiving UDP messages is performed through a socket. After a request is written into the socket the client will be blocked until the reply is returned from the server side. The *generator* and *resolver* objects are used to perform bindings at the server and client sides respectively. The *generator* maps a target interface to a name that contains specific information to enable clients to locate the target interface. In contrast, the *resolver* converts the name into a proxy-object (stub) along with related communication stack to represent the exported interfaces. This is how the binding mechanism works in FlexiNet. The "Binder" term usually refers to an object that can generate and resolve names. FlexiNet can support multiple binders and protocols for a single process. This opens the door for choosing the appropriate binder for performing the bindings. Each protocol stack may associate with many binders that can be selected dynamically. Figure 7 presents two different basic binders named *Green* and *Red*. The *Green* binder uses a protocol that is based on Rex over UDP whereas the *Red* binder uses IIOP over TCP. The *Choice* binder selects which binder to use based on the interface type and name being named or resolved.

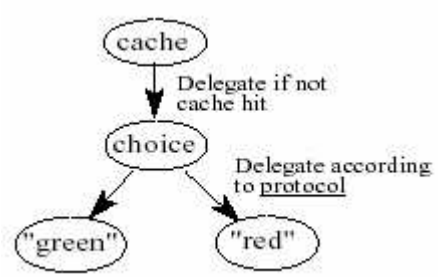

Figure 7: A hierarchy of binders [11]

FlexiNet achieves reflection by implementing method reflection. Typically method calls to a target object are intercepted and the parameters of method calls are forwarded to the *meta_before* method. There is a possibility of applying some modification at this method. When a reply is received the *meta_after* method is called to apply meta transformations. Figure 8 illustrates this mechanism. The advantage of this approach is that only one stub is needed to apply the functionality of reflection and remote invocation.

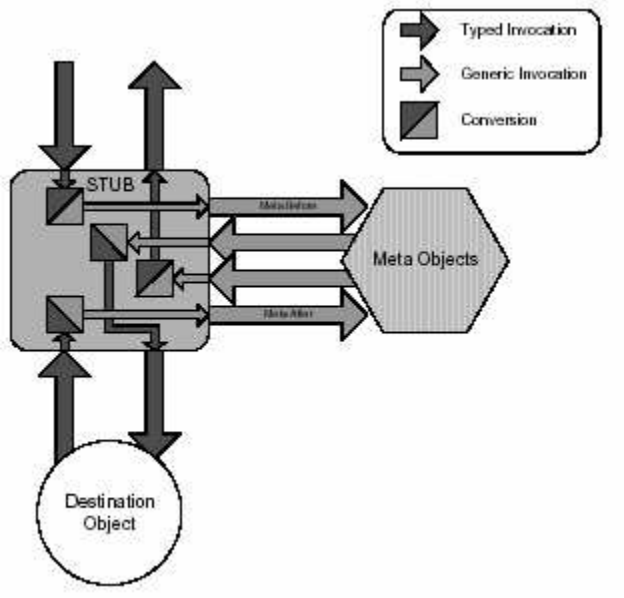

Figure 8: Reflection [11]

FlexiNet is a Java-based framework that is aimed at modularity and reusability rather than performance. The modularity easily allows management policies to be plugged in as a trade-off between performance and resource usage. The FlexiNet framework performance was tested against RMI and OrbixWeb. Even though its performance was lower, FlexiNet is 100% pure Java and does not depend on external tools to perform invocation. In contrast, RMI and OrbixWeb respectively depend on native method and stub compilation tools to function. In FlexiNet, a synchronization model is used for the communication. This type of communication model is characterized by poor utilization of bandwidth.

4.1.5 Globe

The Globe distributed system [13] is a middleware platform particularly constructed for dealing with large-scale systems. The principle of reflection is achieved by supporting replication, caching and distribution of object's state. The Globe architecture is based on a special kind of object model called *distributed shared object (DSO)*. A distributed shared object can be viewed as a conceptual object that is physically distributed over many machines. Each object is also viewed as a wrapper that encapsulates all the object's strategy for replication and distribution. This allows different application objects to encompass policies that fit their requirements. The communication between all processes is done through distributed shared objects. Hence, a large number of processes may share a single object. A Globe object usually provides one or multiple interfaces that define a set of methods. Processes can interact with each other by invoking these methods. The transparency aspect of distributed objects is achieved by hiding all method implementations from client process behind the interfaces. When a process fires a call to an object's method, binding to that object will take place. The binding can be done at one of the object's contact points. An interface along with its implementation is the result of the binding process. This interface belongs to the DSO and is placed at the client's address space. The resulting implementation is called a *local representative object*. This local object is a representative of the DSO, taking care of all implementations (i.e. replication, communication protocol, and state distribution). Figure 9

illustrates a Globe object that is distributed across four address spaces.

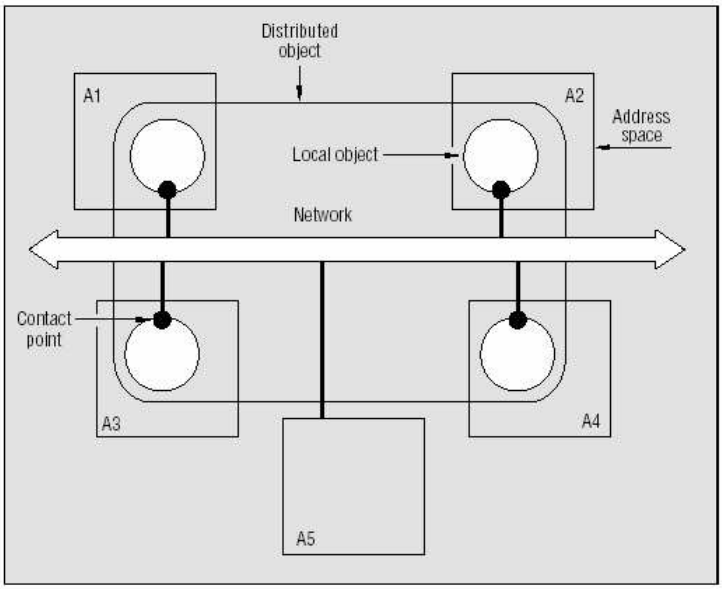

Figure 9: Globe object model [13]

The above figure shows a distributed shared object (DSO) distributed across four address spaces. At each address space there is a local object that represents the DSO. The local objects can communicate with each other even though they reside in different address spaces. Each local object is a self-contained composite object that minimally consists of four subobjects as shown in Figure 10.

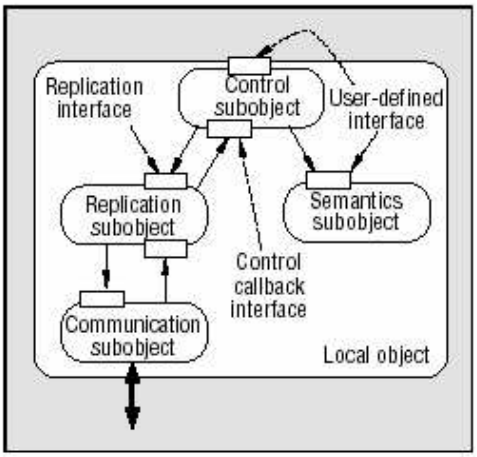

Figure 10: A distributed object's general implementation [13]

Semantics subobject: implements part of the actual distributed object semantic, which consists of user defined primitive objects written in programming languages (i.e., C, C++, and Java).

Communication subobject: handles the communication between the distributed objects residing on different machines. The communication subobject may provide different types of communication (i.e., point-to-point, multicast, or both).

Replication subobject: keeps the global state of the distributed object consistent among all local objects based on a given coherence strategy. Different replication subobjects may have different replication algorithms.

Control subobject: takes care of method invocations from a process and controls the communication between the semantics and replication subobjects.

To communicate through a distributed shared object, a client process requires to first bind to that object. There are two different phases for achieving binding. Finding a distributed object is the first phase, which consists of name and location look-up steps. A client process has to send the object's name to a naming service. An object handle then is sent back by the naming service. This handle is basically a worldwide unique location-independent object identifier (OID) that is used to identify each DSO. This identifier thus can be passed between processes as an object reference. Finally, the OID is mapped to one or more contact addresses that are obtained from a location service. Installing a local object is the second phase, which consists of selecting the appropriate contact address and an interface implementation. A suitable contact address that describes where and how the DSO can be reached needs to be selected. The address selection can be done based on different criteria that determine preference of one address over another. Then the local run-time system generates a new local object in the client's address space and associates the new representative with the DSO. Figure 11 describes the binding steps.

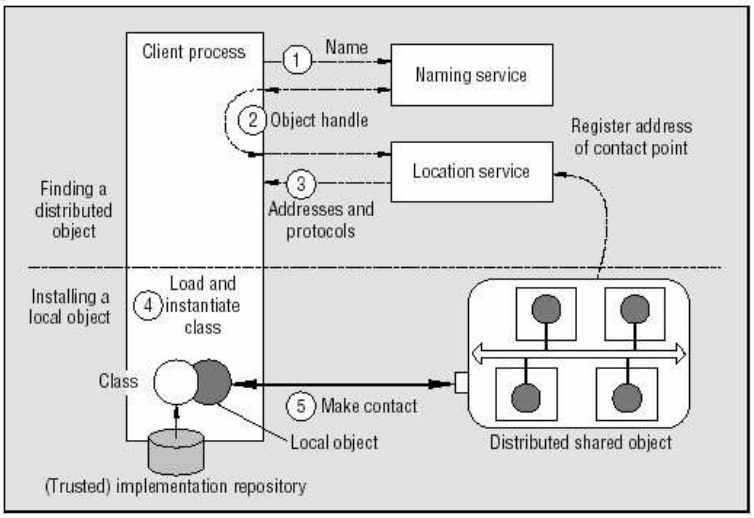

Figure 11: Binding a process to a distributed shared object [13]

4.2 Tuple Space Middleware

Communication in a wireless environment is characterized by frequent disconnections and limited bandwidth. Communication models such as message passing, RPC, or RMI all have the drawback of tight coupling. This means that a sender has to know the exact identity and address of a receiver. Also, the sender has to wait for the receiver to be ready for exchanging information (synchronization paradigm). In distributed open systems this tends to be too restrictive. A decoupled and opportunistic style of computing is thus required. Computing is expected to proceed even in the presence of disconnection and to exploit connectivity

whenever it becomes available. One solution is the concept of tuple space, which was initially introduced by Gelernter in [23] as part of the Linda coordination language. Tuple Space systems have proved their ability for facilitating communication in wireless settings. In general, a tuple space is a globally shared, associatively addressed memory space that is used by processes to communicate. A tuple space system can be realized as a repository of tuples, which are basically a vector of typed values or fields. Client processes create tuples and place them in the tuple space using a write operation. Also, they can concurrently access tuples using read or take operations. Most tuple space systems support both versions of the tuple retrieval operations, blocking and non-blocking. A template, which is similar to a tuple, is used to match the contents of tuples in the tuple space during the retrieval operations. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple field. This form of communication fits well in mobile setting where logical and physical mobility is involved. We review here some tuplespace-based systems that are based on Linda.

4.2.1 LIME (Linda in a Mobile Environment)

The LIME model [14] is designed to provide application designers and developers with a coordination layer that can deal with logical or physical mobility. LIME inherits and adapts the communication model proposed by Linda [15]. The coordination aspect in Linda is accomplished by using a tuple space that can globally be shared by all mobile units. The tuple space can be accessed through a basic set of Linda primitives that allow inserting, reading, or withdrawing tuples. Processes interact with each other via a shared *tuple space*, which is basically realized as a repository of *tuples*. Multiple processes can simultaneously access these tuples. A tuple is a data structure that has an ordered sequence of data types. Tuples can be inserted, deleted, and read by using *out*(t), *in*(p), and *rd* operations respectively. The *in* and *rd* operations are implemented using synchronous model. This means the processes performing one of these operations are suspended until a matching tuple becomes available in the tuple space. The communication aspect offered by Linda does not require both the sender and receiver to be available at the same time. Also, there is no need for advance knowledge of the participant's location for data exchange. However, the idea of a static, persistent, and global visible tuple space that is assumed by Linda is not reasonable to meet the mobility demands. The global context is formed by a transient community of mobile hosts (and mobile agents). Each component contributes its own individual context. Naturally the global context needs to be changed whenever the transient communities are dynamically changed due to connectivity variation or agent migrations. LIME adapts the Linda model via the notion of a *transiently shared tuple space* that shifts a fixed context to a dynamically changing one. Basically Linda tuple space is divided into many tuple spaces and each one is permanently attached to mobile hosts or agents. Transient sharing accordingly allows dynamic reconfigurablity of the tuples contents. As shown in Figure 12, each mobile unit can only access the global context via an interface tuple space (ITS) that is permanently attached to the unit itself and travels with it when migration happens.

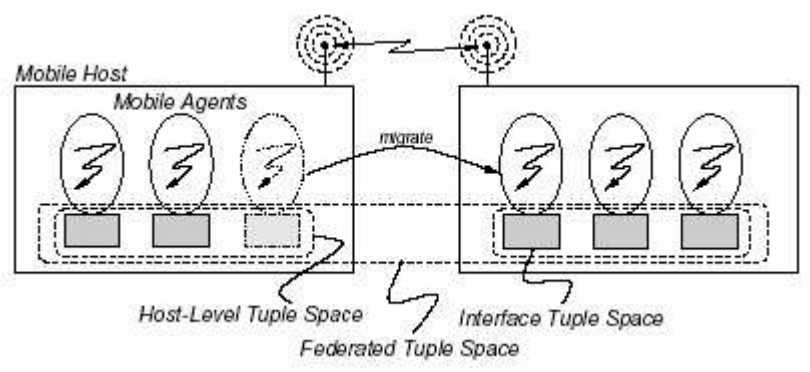

Figure 12: Transiently shared tuple spaces in LIME [15]

The ITS contains tuples that can be shared with other mobile units. It is accessed using Linda operations (*in*, *out*, and *rd*). When a new mobile unit arrives to the tuple space, the contents of its ITS are engaged with those of other mobile units. The resulting set of tuples are made accessible through the ITS of each unit after it has been dynamically recomputed. Similarly, when a mobile unit departs the tuple space, the contents are disengaged from the tuple space. Mobile agents are the only active units that carry a "concrete" tuple space with them. Transient sharing gives a mobile host the illusion of a local tuple space that holds all the ITSs belonging to multiple agents co-located on its machine. This local tuple space is known as a *host-level tuple space*. The host-level tuple space can also be transiently shared among several connected hosts, forming the *federated tuple space*.

LIME supports coordination that hides the details of distribution and mobility. This relieves designers from dealing with these changes; however, some mobile applications demand to deal with the data distribution explicitly. LIME provides this kind of control by extending Linda operations with tuple location parameters. The *out* operation is extended by adding the location parameter *out*[λ] that represents the agent or host identifier responsible for holding the tuple. The new semantics of *out*[λ](*t*) is equivalent to two steps. The first one *out*(*t*) inserts the tuple *t* in the ITS of the agent invoking the operation, ω. The tuple t has two locations, a current location ω and a destination location λ. Whenever the agent λ is connected the tuple *t* is moved to the destination location. Otherwise, the tuple *t* remains at the current location ω. The location parameter notion is also available for the *in* and *rd* operations. To achieve full context awareness, LIME exposes the necessary information via a read-only, system-maintained tuple space, called *LimeSystem*. The tuples of LimeSystem hold information about the mobile units that exist in the community, and their relationship. LIME also extends the basic Linda tuple space with a notion of *reaction*. After each operation on the tuple space, a reaction R(*s*, *p*) is chosen and the pattern *p* is compared against the content of the tuple space. If a matching tuple is found, *s* is executed; otherwise the reaction is a skip.

LIME provides some form of context-awareness but the overhead cost is high. The blocking behaviour of LIME primitives adds to the overhead. There is no support for behaviour adaptation.

4.2.2 TSpaces

TSpaces [16] is a network middleware platform, which is based on the combination of the tuple space idea and database technologies. It provides a lightweight database, an extensible

computation environment, and a secure communication layer. Moreover, being implemented in Java, it adds the portability feature to the system. TSpaces expands the Linda [15] framework with real data management and the capability of downloading new data types and functionality dynamically. TSpaces contains several features that are not available within other tuple space based systems. TSpaces supports both nonblocking and blocking versions of the tuple operators (*read* and *take*). New data types and operators can be dynamically defined and downloaded into a TSpaces server and used immediately. TSpaces operations are performed in a transactional context to ensure data integrity. TSpaces provides an indexing mechanism and a query capability to facilitate data retrieval. Data and operations are fully decoupled in order to add or remove operations without affecting the database. The TSpaces client and server communicate with each other through tuples. The tuple is basically an ordered field that describes a type and a value. The TSpaces server may accommodate several tuple spaces. Once the tuple spaces are created on the server side, the TSpaces client may perform a variety of operations. Figure 13 illustrate a view of two clients communicating through a TSpaces server.

The TSpaces server forms a centralized model that listens and responds to client's requests. As shown in Figure 13, client 1 triggers a write operation to write the <test1> tuple into the tuple space. The <test1> tuple then is stored in the TSpaces tuple database. Client 2 triggers a read operation, indicating <test1> as the required tuple. The request is sent to the server to search for <test1> in the tuple database. When the <test1> tuple is found, a copy of the tuple is passed to the client 2. Since it is a read operation and not a take operation a copy of <test1> will remain in the tuple database. There are two types of server system tuple spaces, Galaxy and Admin. The Galaxy space contains tuples describing each tuple space that exists on a TSpaces server. The Galaxy tuple space is the start point of all operations. It implements the *CreateTuplespace*, *DestroyTuplespace*, and *TuplespaceExists* operations. The Admin tuple space contains the access control permissions for each tuple space, the groups that each client belongs to, and the user name and password of each client. It is primarily used to check whether the issuer of each operation has the proper access control privileges.

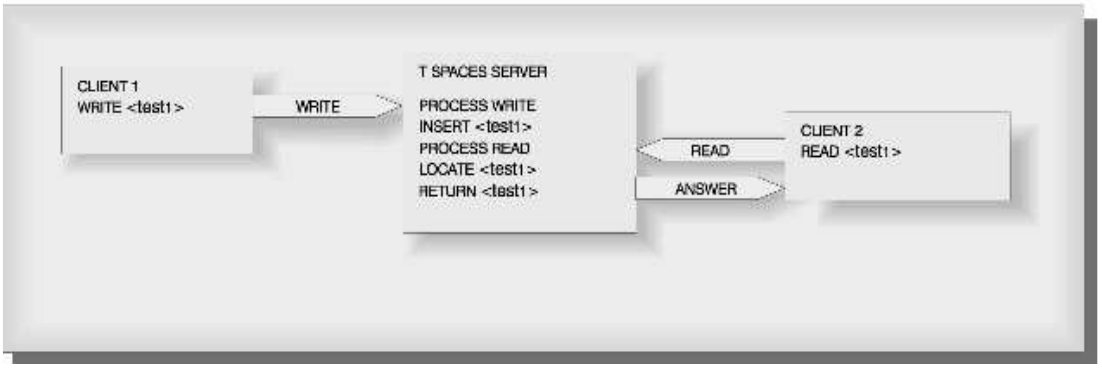

Figure 13: TSpaces overview [16]

TSpaces does not support any kind of replication that can improve the system performance. Usually there is one server per TSpaces. The system performance will be affected if the server is overloaded or crashed. Clients need some kind of caching technique to overcome the disconnections issue. Due to sudden disconnections, clients cannot access any information that retains on the TSpaces server. Using an efficient caching mechanism may resolve this

problem. Also executing transactional operations on the tuple space database introduces high costs in terms of delay time and consumes much bandwidth. It is not clear if the write operation performs any checks to avoid tuple redundancy.

4.2.3 JavaSpaces

JavaSpaces [17] is an attractive technology for dynamic communication, coordination, and sharing objects across a distributed computing environment. The JavaSpaces package provides a management structure or a virtual space for shared objects to exist and be accessed by client processes. This enables providers and requesters to exchange tasks, requests and information in form of Java objects. The JavaSpace implementation provides developers with useful tools and Java style methods that support tuple space operations (i.e., read, write, take) to create and store objects with persistence. The JavaSpaces system is inspired by Linda, but differentiates itself from Linda by using objects to represent tuples. Moreover, each field in the tuple is an object that may have some methods. This allows objects expressed in Java classes to communicate as a tuple and be persistent. Figure 14 shows the main ideas for a JavaSpaces technology.

As discussed in TSpaces, a space is hosted by a server computer and manages objects placed into it by client processes. The server side is assumed to be a powerful machine that may run a relational or object-oriented database to achieve persistence of tuple space.

The JavaSpaces architecture supports a mechanism to implement transactions on compound objects and multi-operations. All objects that will be managed by the space need to implement the *Entry* interface. The client process then can write *Entry* objects to a space, read a copy of the objects, and take objects from the space. *Entry* objects need to be matched by read or take operations using a pattern with matching data field. Objects can be shared using asynchronous communications object with buffering and persistence automatically managed.

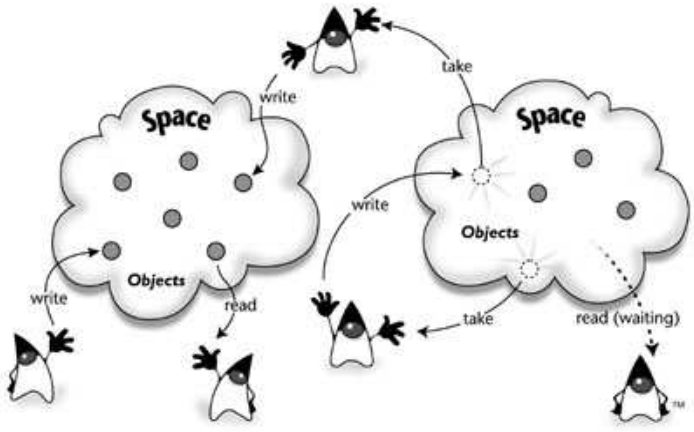

Figure 14: JavaSpaces technology [17]

In general, tuple space based systems do not support any notion of context awareness, which is considered an important element for mobile environments. The data structures used to represent tuples hosted by the server do not provide context information to applications. The current primitives do not allow applications to influence the middleware behaviour. More complex primitives are needed to support the notion of adaptation. Also, currently there is no provision for a reconciliation mechanism among tuples. This is an important requirement

since there is a possibility of tuple duplication in tuple-space based systems. Different clients may obtain a copy of a tuple and keep updating it while they are disconnected. Upon reconnection, the tuple space system should check whether several hosts share the same tuple. If so, a reconciliation mechanism needs to be initiated to reflect the updates due to all clients.

4.3 Context-Aware Middleware

Mobile systems run in an extremely dynamic environment. The execution context changes frequently due to the user's mobility. Mobile hosts often roam around different areas, and services that are available before disconnecting may not be available after reconnecting. Also, the bandwidth and connectivity quality may quickly alter based on the mobile host movements and their locations. The application developers cannot predict all the possible execution contexts that allow the application to know how to react in every scenario. The middleware has to expose the context information to the application to make it aware of the dynamic changes in execution environment. The application then instructs the middleware on how to adapt its own behaviour in order to achieve the best quality of service. Context-aware computing was first proposed by [24]. Many research groups gave special attention in particular to location awareness. For example, location information was exploited to provide travellers directional guidance [25], to discover neighbouring services [26], and to broadcast messages to users in a specific area [27]. Most location-aware systems depend on the underlying network operating system to obtain location information and generate a suitable format to be used by the system. The heterogeneity of coordination information is not supported and hence different positioning systems are required to deal with different sensor technologies, such as the Global Positioning System (GPS) outdoors, and infrared and radio frequency indoors. We briefly review one middleware system that supports different positioning systems through a common interface.

The Nexus [18] middleware is designed to be a generic platform for all kinds of location-aware applications. The platform consists of four components that need to cooperate: the user interface, the sensor systems, the communication and the data management. Figure 15 shows the architecture of the Nexus platform.

The user interface runs on the mobile device carried by users and enables the Nexus clients or location-aware applications to communicate with the Nexus platform. It has to adapt to a variety of Nexus stations, related to the diversity of computing power, memory size, network connection or displays. Also, the user interface is platform independent.

The sensor systems are required to provide positioning information to the Nexus system. A set of sensors can be used in the system since the Nexus applications may run in an outdoor or indoor area. It would be difficult to use only one sensor for positioning in both environments. For example, Global Positioning System (GPS) can be used outdoors but not indoors as its satellite signals are blocked by buildings. It is possible to combine two or more measurements from different sensors to increase the accuracy. Thus, there is real need of a multi-sensor tool to work in a verity of environments with high accuracy.

The communication unit takes care of data exchange between the different components of the platform. Usually, the mobile devices use different types of network connections to access the necessary information. The major task of this unit is to decide which network component will do the work best. The communication unit also has to support the adaptivity of the system in case of bandwidth fluctuation.

The data management is responsible for different aspects concerning the management of data. It organizes the data in a distributed environment and enables the processing to be shared among different servers to reduce the response time. The spatial data also needs to be offered in multiple representations to meet the needs of different applications. The interoperability of data must be guaranteed.

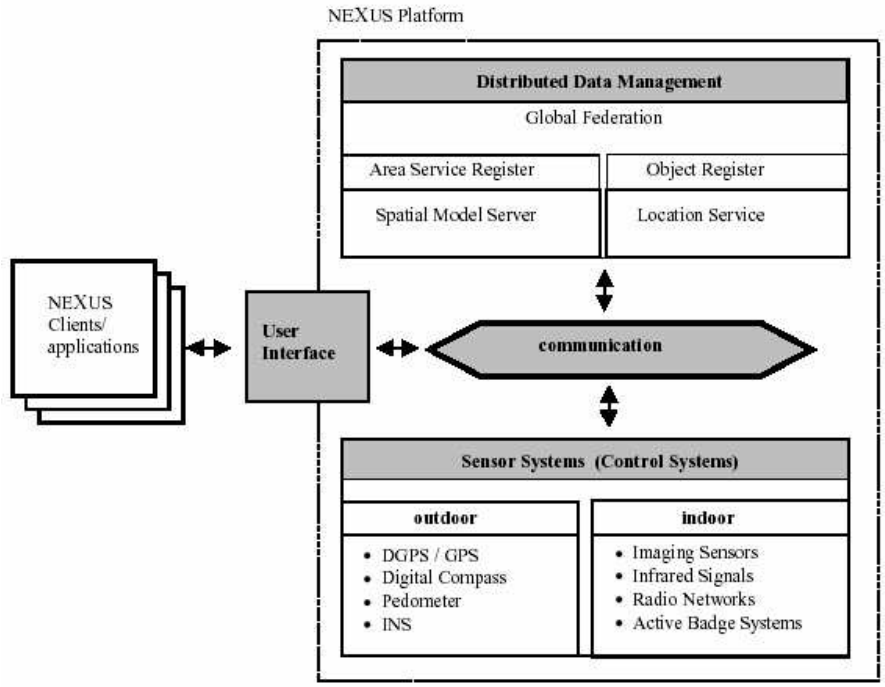

Figure 15: Architecture of the Nexus platform [18]

This system supports only one type of context awareness, which is location awareness. It is necessary to take into account other types of context awareness such as internal resources (i.e., memory size, battery and processor power) or external resources (i.e., network bandwidth and connectivity quality). However, making the application aware of the execution context adds an additional level of complexity to the application developers. More efforts need to be directed towards an easy context representation and simple interfaces that enables the applications to interact with the underlying middleware.

4.4 Event-Based Middleware

Invocation-based middleware systems such as CORBA or Java RMI are useful abstractions for building distributed systems. The communication model for these platforms is based on a request/reply pattern: an object remains passive until a principle performs an operation on it. This kind of model is adequate for a local area network (LAN) with a small number of clients and servers, but it does not scale well to large networks like the Internet. The main reason is that the request/reply model only supports one-to-one communication and imposes a tight coupling between the involved participants because of the synchronous paradigm. This model is also unsuitable for unreliable and dynamic environment. The event-based communication paradigm [28] is a possible alternative for dealing with large-scale systems. Event notification is the basic communication paradigm that is used by event-based middleware systems. Events contain data that describes a request or message. They are propagated from the sending

components to the receiver components. In order to receive events, clients (*subscribers*) have to express (*subscribe*) their interest in receiving particular events. Once clients have subscribed, servers (*publishers*) publish events, which will be sent to all interested subscribers. This paradigm hence offers a decoupled, many-to-many communication model between clients and servers. Asynchronous notification of events is also supported naturally. There are several examples of middleware based on the event-based systems, but not limited to, Hermes [29], CEA [28], STEAM [30], JEDI [35] and ToPSS [46]. We discuss these examples briefly.

Hermes

Hermes [29] is a distributed event-based middleware architecture that is powerful enough to support many large-scale distributed applications. The system architecture consists of two main components, *event clients* and *event brokers*. The event clients represent both *event publishers/subscribers* and communicate with the event brokers by using asynchronous message-passing (XML). The event brokers contain the entire functionality of the middleware used by the event clients. This allows the event clients to be lightweight components and hence can be run on mobile systems. The main role of the brokers is to receive subscriptions from subscribers and then use a content-based routing algorithm [31] to deliver events from publishers to all interested subscribers. To consume less bandwidth and increase scalability, event stream filtration is done as close as possible to the event publisher. Figure 16 shows a distributed application built on top of Hermes.

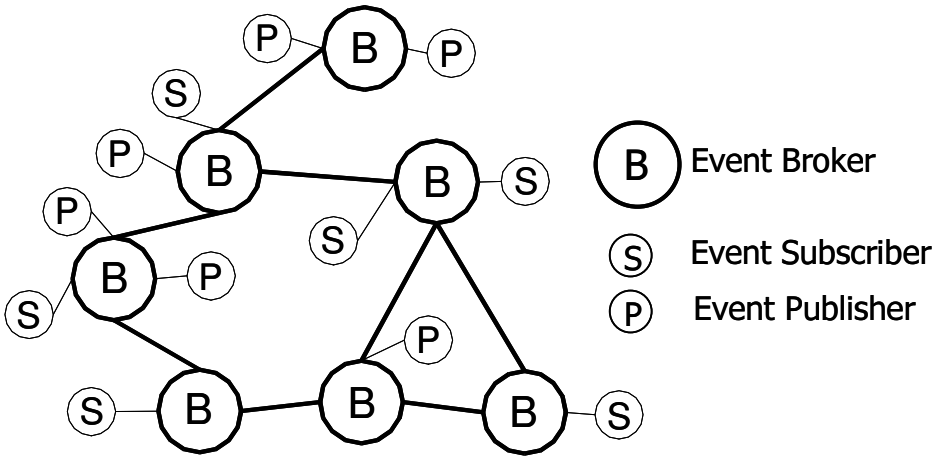

Figure 16: A distributed application built with Hermes [29]

Hermes depends on the service of a peer-to-peer overlay routing network that is similar to [32]. The overlay routing network provides an abstraction for routing, the *route (message, destination_id)* function. This allows a broker to send a message to another broker, which is then routed via the overlay network. Several advantages are gained from using the overlay network for event dissemination. No global state knowledge is required for event routing. This means that an event broker does not need to have advance knowledge of all the subscriptions or event sources. System robustness is increased because the overlay network transparently deals with link or node failures. The overlay routing operation is used to set up

rendezvous nodes [33], which are special event brokers that are well known to both publishers and subscribers. They are responsible for creating event dissemination trees [34]. In Hermes, all events are part of an *event type* hierarchy that contains *event attributes*. Before an event client subscribes, it first needs to specify an event type it is interest in. Then it provides a *filtering expression* that operates on the event attributes supplied by this type. Since the event type and its attributes are known to the middleware, it is simple to type-check events and subscriptions at runtime and to notify the user about any mismatch.

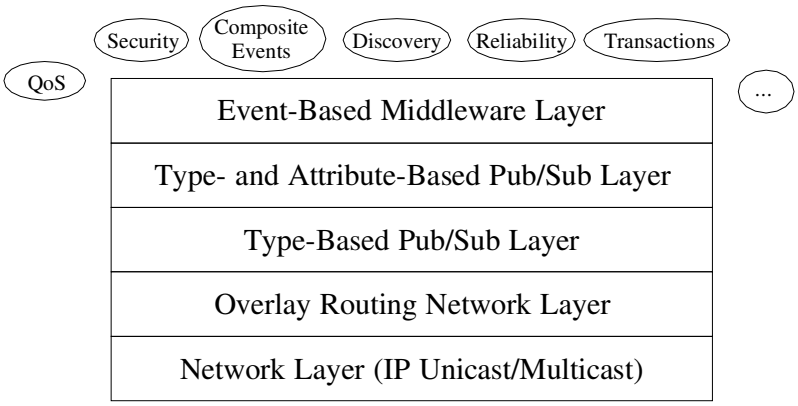

Figure 17: The layered architecture of Hermes [29]

Hermes' architecture is represented in a layered stack as illustrated in Figure 17. The *overlay routing layer* offers a basic communication service between the event brokers. The functionality of publish and subscribe is split into two *pub/sub layers* that implement the event dissemination algorithms with source-side filtering. Middleware services (i.e., QoS, security, and reliability) are implemented in the form of modules on top of the *event-based middleware layer*.

Hermes does not support the notion of composite events. The architecture can be extended by adding composite event services that combine primitive events into composite events. The composite event services register their clients' interest with appropriate event resources and notify clients of composite events. A special mechanism is needed to model event arrival from different sources and to specify composite events. This however may complicate the system architecture and add extra cost.

The Hermes event broker does not provide a persistent storage where events can be stored for the provision of a reliable event service. Storing events can guarantee the correct delivery of events to the interested subscribers even during the brokers' failures. Also, brokers should provide a mechanism that prevents a mobile user from missing events of interest while disconnected from the networked systems.

4.5 Other Middleware Solutions

There are many other middleware solutions that have been proposed particularly to target mobility aspects. Unpredictable disconnections are one of the major mobility issues that have been addressed by several systems. Systems like Coda [36], its successor Odyssey [37], Bayou [38], [39] and xmiddle [40] have used data replication to increase data availability to mobile users. This allows users access to replicas and to continue their tasks whenever the

disconnection operations take place. Each system uses different mechanisms to guarantee the ultimate consistency among the replicas. These mechanisms include the support for discovery of inconsistent data as well as data reconciliation.

Services discovery is another well-know problem introduced by user mobility. In a static environment, new services can be easily discovered by asking service providers to register with a well-known location service. In a mobile computing environment, the situation is different since mobile hosts often roam around various areas. Services that were present before disconnecting from the network may not exist after reconnecting. Jini [41] and Ninja *Service Discovery Service* (SDS) [43] are examples of systems that support dynamic service discovery. In this section, we describe the main architecture and features of two systems which support disconnected operations (Bayou) and discovery of services (Jini).

4.5.1 Bayou

The Bayou system [38], [39] is a platform that provides replicated, highly-available, variable-consistency, and mobile storage for building collaborative applications. It also maintains conflicts caused by concurrent activity while relying only on weak connectivity for mobile computing. Replication is required in order to allow disconnected users to access a common storage. The Bayou system enables users to read from and write to any copy of the database. Communicating with several replicas may not be feasible all the time, which affects the correctness of performing write or update operations to all other replicas. This leads to weakly consistent replicated data that is not transparent to applications. The applications should be aware of the weakly consistent data since they must be involved in solving conflicts. The Bayou system provides support for application-specific conflict detection and reconciliation. A conflict can be detected when performing a write operation that not only includes the data but also a *dependency set*. It is a collection of queries and their expected results that are supplied by an application. A conflict is detected after the queries run on a copy of a database and do not return the expected results. Bayou also provides means for resolving such conflicts. A write operation also includes a procedure called *mergeproc* that is called whenever a write conflict is detected. Mergeprocs are mobile agents that clients create and pass to servers where they run to resolve conflicts. The Bayou system ensures that all copies of a database are converging towards identical states. Servers propagate write operations among database copies using an *anti-entropy* protocol [42]. This protocol is adopted to guarantee that any two machines will be able to propagate updates between themselves. Even machines that never directly communicate can exchange updates via intermediaries. Each server periodically selects another server with which to perform a pair-wise exchange of writes. At the end, both servers have identical copies of the database. Eventual consistency is reached by having the same write operations applied in the same order on both servers. Figure 18 illustrates the main components of the Bayou architecture.

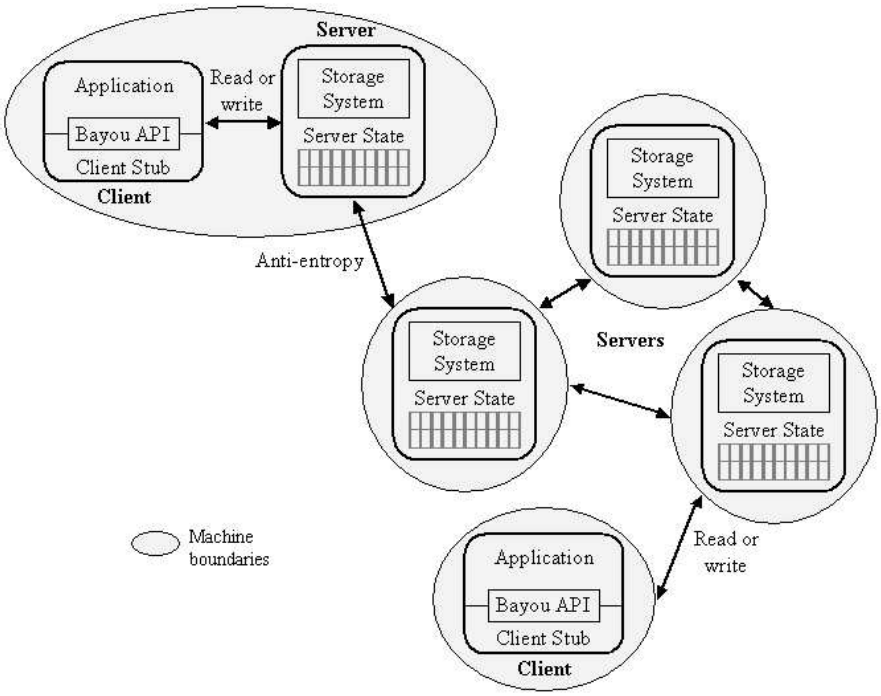

Figure 18: Bayou System Model [39]

The Bayou system uses application knowledge for dependency checks and merge procedures to achieve conflict detection and resolution, unlike Coda [36], which promotes transparency of conflict detection and resolution. Bayou also allows replicas to always remain accessible. This permits clients to continue to read previously written data and to issue new writes, but it may lead to cascading conflict resolution if the newly issued operations depend on data that are in conflict. As shown in Figure 18, a client and a server may be co-resident on a host (i.e., laptop or PDA). This enables lightweight servers to run on portable machines. The main drawback of the Bayou system is the client-server model of its architecture. Their model uses RPC protocol for communication, which follows the synchronous paradigm. Another disadvantage is that the complete data collection needs to be replicated in full on a number of servers. This is inadequate for handheld devices due to their capacity limitation. Hence handheld devices can only play the role of clients in this architecture. Partial replicas that contain subsets of a data collection, which is important for some portable devices, need to be supported.

4.5.2 Jini

Mobility poses a very interesting problem to the end mobile users: how to locate a particular network service or device out of many available services and devices. The key challenge for these users will be *discovering* the most appropriate service for a certain task among many accessible services. The word “appropriate” depends on a user-specific definition such as cost, location, and accessibility. The concept of service is defined as an entity that can be used by a user, an application or another service. A service can be in form of a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user. One example of such a service is controlling the lights and the doors in a house. Accessing such services in a secure manner is also a crucial requirement. Clients should not

be expected to keep track of running services or knowing which ones can be trusted. Jini [41], a middleware system, provides dynamic discovery of services that is built on the idea of *federating* groups of devices and software components into a single, dynamic, distributed system. The main aim of this federation is to increase the flexibility of networks, the simplicity of access, ease of administration, and support for sharing. The system is focusing on making the network a more dynamic entity by enabling services to be found, added, and removed from a federation in a flexible manner. In Jini, services can be found and resolved using a lookup service, which maps a set of interfaces offered by a service to a collection of objects that use the service. Two protocols, *discovery* and *join*, are used to add a new service to a lookup service. The service provider first uses the discovery protocol to find an appropriate lookup service, and then registers a service object (proxy) and its service attributes with the lookup service using the join protocol. When a client requests a service, a copy of the service object is moved to the client and used by the client to talk to the service. Finally, the client will be able to interact directly with the service provider via the service object (proxy). Lookup happens whenever a client needs to find and call a service that is described by its own interface type. Figure 19 describes the lookup process.

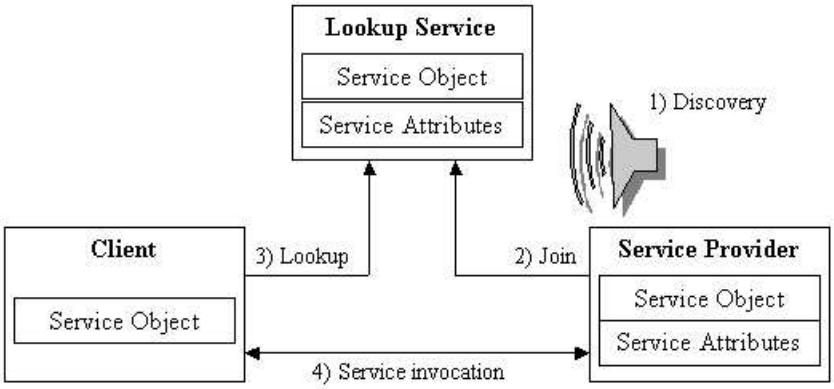

Figure 19: Lookup process in the Jini system

Services in Jini system can be accessed in *lease* manner, which gives more guarantees to access services over a time period. A service lease is negotiated between the user and the provider of the service. Each lease needs to be renewed before it is freed; otherwise the client or network fails. Exclusive leases prevent anyone else from concurrently using the same resource. Nonexclusive leases allow users to share a resource. The Jini system supports the notion of transactions where a number of operations belonging to a single or multiple services can be wrapped in one transaction. It also supports distributed event-based programs, which have been discussed in the previous section. A distributed security model is also offered to provide access to resources to authorized users only.

The Jini system relies on the availability of high network bandwidth, which enables services, devices, and users to join and leave a network in a transparent manner. It also assumes that devices have some memory and processing power and the network latency is reasonable. This does not exist in a mobile computing environment where there is limited bandwidth and devices with scarce resources. The Jini infrastructure is Java based which simplifies the Jini architecture, as the Java programming language is the implementation language for components. This hence adds the ability to dynamically download and run codes on any

platform to the features of the Jini architecture. However, the Jini architecture relies on the Java application environment rather than on the Java programming language. This implies that any programming language can be supported by a Jini system as long as it has a compiler that generates compliant bytecodes for the Java programming language. Java Remote Method Invocation (RMI) is used to support communication between services. RMI follows the synchronous communication model, which blocks a requester until the returned result is received by the requester.

5 Discussion

This section summarizes the previous discussion on next-generation middleware with an emphasis on lessons learned from investigating the proposed solutions presented in the previous section. We particularly aim to highlight in which extent these solutions are suitable for mobile settings.

It is a major challenge to solve all problems of mobile distributed systems. This is true due to the high degree of dynamism in mobile environments. Current middleware platforms like CORBA cannot successfully run in such an environment. Hence, there is an urgent need for new solutions that support particular application requirements such as dynamic reconfiguration, context-awareness, and adaptation.

We believe that the reflective approach described in Section 4.1 provides a solid base for building next generation middleware platforms and overcomes the limitations of the current middleware technologies. More specifically, the architecture follows a white box philosophy that provides principled and comprehensive access to internal details. It can also decrease problems of maintaining integrity since each object/interface is attached to a single meta-object at a time. Therefore, any modification to a meta-object can only affect a single object. Some reflective systems support higher level of reflection since they can add or remove methods from objects and classes dynamically and even alter the class of an object at run time. In contrast, others concentrate on a simpler reflective paradigm to achieve a better performance. Their reflective mechanisms are not part of the usual flow of control and only invoked when required. Reflective middleware like FlexiNet [11] and DynamicTAO [8] are built around the concept of object-oriented and component frameworks respectively. Component Frameworks (CFs) were initially defined by Szyperski [47] as "*collection of rules and interfaces that govern the interaction of a set of components plugged into them*" There are several advantages of using CFs over the object-oriented approach. The uses of CFs are not limited to a particular programming language and there is no inheritance relation between components and framework. Hence, components and CFs can be developed independently, distributed in binary form, and combined at run time. We have noticed that the issue of consistent dynamic reconfiguration is still under research. There is some work in this area that has focused on developing reconfiguration models and algorithms that enforce well-defined consistency rules while minimizing system disturbance [48]. Performance is another issue that remains a matter for further investigation. All of the reflective systems presented previously impose a heavy computational load that would cause significant performance degradation on mobile devices.

Tuple-space systems exploit the decoupled nature of tuple spaces for supporting disconnected operations in a natural manner. By default they offer an asynchronous interaction paradigm

that appears to be more appropriate for dealing with intermittent connection of mobile devices, as is often the case when a server is not in reach or a mobile client requires to voluntary disconnect to save battery and bandwidth. By using a tuple-space approach, we can decouple the client and server components in time and space. In other words, they do not need to be connected at the same time and in the same place. Tuple-space systems support the concept of a space of spaces that offers the ability to join objects into appropriate spaces for ease of access. This opens up the possibility of constructing a dynamic super space environment to allow participating spaces to join or leave at arbitrary time. The ability to use multiple spaces will elevate the overall throughput of the system. Throughout our study, we have noticed that JaveSpaces [17] and TSpaces [16] typically require at least 60Mbytes of RAM. This is not affordable by most handheld devices available on the market nowadays.

Context-Aware systems provide mobile applications with the necessary knowledge about the execution context in order to allow applications to adapt to dynamic changes in mobile host and network condition. The execution context includes but is not limited to: mobile user location, mobile device characteristics, network condition, and user activity (i.e., driving or sitting in a room). The context information is typically disclosed in a convenient format to the applications that instruct the middleware system to apply a certain adaptation policy. To our knowledge, most context-aware applications are only focusing on a user's location while other things of interest are also mobile and changing. We believe that a reflective approach may improve the development of context-aware services and applications. In general, a reflective system provides mobile applications with context information that they need to optimise middleware and their own behaviours. One reflection solution [49] has suggested the use of metadata and reflection to support context-aware applications.

Traditional, invocation-based middleware like CORBA follow a request/reply communication style, which does not scale well to large networks like the Internet. Event-based paradigms present an interesting style that supports the development of large-scale distributed systems. In such a system, clients first announce their interest in receiving specific events and then servers broadcast events to all interested clients. Hence, the event-based model achieves a highly decoupled system and many-to-many interaction style between clients and servers. We believe that not a lot of work has managed to merge the publish/subscribe communication approach with event-based middleware systems. Most existing systems do not combine traditional middleware functionality (i.e., security, QoS, transactions, reliability, access control, etc.) with the event-based paradigm. We feel that event-based middleware can be more successful if such functionality is provided in the future. Event-based systems also do not integrate well with object-oriented programming languages due to the major mismatch between the concept of objects and events. Events are viewed as untyped collection of data (attribute/value pairs) whereas current programming languages only support typed objects. Hence, events should support data typing in order to be treated as objects. In addition, the developers are responsible for handling the low-level event transmission issues. Current publish/subscribe systems are restricted to certain application scenarios such as instant messaging and stock quote dissemination. This indicates that such systems are not designed as general middleware platforms.

From this discussion, we can realize that until this moment there is no middleware system that can fully support the requirements for mobile applications. Several solutions have considered one aspect or another; however, the door for further research is still wide open.

6 Conclusions

The proliferation and development of wireless technologies and portable appliances have paved the way for a new computing paradigm called mobile computing. Mobile computing software is expected to operate in environments that are highly dynamic with respect to resource availability and network connectivity. Traditional middleware products, like CORBA and Java RMI, are based on the assumptions that applications in distributed systems will run in a static environment; hence, they fail to provide the appropriate support for mobile applications. This gives a strong incentive to many researchers to develop modern middleware that supports and facilitates the implementation of mobile applications.

We discussed the state-of-the-art of middleware for mobile computing. We presented common characteristics and a set of requirements for mobile computing middleware, which allows us to better understand the relationship between the existing bodies of work on next-generation middleware. We explained the reasons behind the failure of traditional middleware systems for supporting mobile settings. We also identified, illustrated, and comparatively discussed four middleware classes: reflective middleware, tuple space, context-aware middleware, and event-based middleware. Beside these four categories, a pool of other middleware solutions has been developed to address specific mobility issues. However, none of these middleware systems support all the requirements highlighted in Section 2. We concluded each category with a simple qualitative evaluation and made a number of observations related to some issues that need further investigations.

7 References

[1] Ledoux, T., "OpenCorba: a Reective Open Broker", In Reflection'99, Volume 1616 of LNCS (Saint-Malo, France, July 1999), pp. 197-214. Springer-Verlag.

[2] RIVARD F., "A New Smalltalk Kernel Allowing Both Explicit and Implicit Metaclass Programming", In Workshop "Extending the Smalltalk Language", OOPSLA'96, San Jose, California, October 1996.

[3] BRANT J., FOOTE B., JOHNSON R., ROBERTS D., "Wrappers to the Rescue", In Proceedings of ECOOP'98, Springer-Verlag, Brussels, Belgium, July 1998.

[4] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, F´abio Costa, Hector Duran Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski, "The Design and Implementation of Open ORB 2", IEEE Distributed Systems Online, 2(6), 2001.

[5] C. Szyperski, "Component Software, Beyond Object-Oriented Programming", ACM Press/Addison-Wesley, 1998.

[6] ISO/IEC, "Open Distributed Processing Reference Model, Part 2: Foundations", ITU-T Rec.X.902 — ISO/IEC 10746-2, ISO/IEC, 1995.

[7] Kiczales, G., des Rivières, J. and Bobrow, D.G., "The Art of the Metaobject Protocol", MIT Press, 1991.

[8] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and Roy H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing

(Middleware'2000), number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[9] Douglas C. Schmidt and Chris Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", IEEE Communications Magazine Special Issue on Design Patterns, 37(4): 54–63, May 1999.

[10] Kon, F. and R.H. Campbell, "Dependence Management in Component-Based Distributed Systems", IEEE Concurrency, 2000. 8(1): p. 26-36.

[11] R. Hayton et. al., "FlexiNet Architecture Report", ANSA Phase III report, February 1999

[12] D. Otway, E. Oskiewicz, "REX: a remote execution protocol for object-oriented distributed applications", IEEE Press 1987.

[13] M. van Steen, P. Homburg, and A. Tanenbaum, "Globe: A Wide-Area Distributed System", IEEE Concurrency, 7(1):70–78, March 1999.

[14] G. Picco, A. Murphy, and G.-C. Roman, "LIME: Linda Meets Mobility", In D. Garlan, editor, Proc. of the 21st Int. Conf. on Software Engineering, pages 368–377, May 1999.

[15] D. Gelernter, "Generative Communication in Linda", ACM Computing Surveys, 7(1):80–112, Jan. 1985.

[16] Wyckoff, P., McLaughry, S. W., Lehman, T. J., and Ford, D. A., "T Spaces", IBM Systems Journal 37, 3, 454-474. 1998.

[17] Waldo, J., "JavaSpaces", specification 1.0. technical report (March 1998), Sun Microsystems.

[18] Fritsch, D., Klinec, D., and Volz, S., "NEXUS Positioning and Data Management Concepts for Location Aware Applications", In Proceedings of the 2nd International Symposium on Telegeoprocessing (Nice-Sophia-Antipolis, France, 2000), pp. 171-184.

[19] Kiczales, G., des Rivieres, J., and Borrow, D., "The Art of the Metaobject Protocol", The MIT Press 1991.

[20] Yokote, Y., "The Apertos Reflective Operating System: The Concept and its Implementation", In Proceedings of OOPSLA'92 (1992), pp. 414-434. ACM Press.

[21] McAffer, J., "Meta-level Architecture Support for Distributed Objects", In Proceedings of Reection'96 (San Francisco, 1996), pp. 39-62.

[22] Smith, B., "Reflection and Semantics in a Procedural Programming Language". Ph.D. thesis (January 1982), MIT.

[23] Gelernter, D., "Generative Communication in Linda", ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, pp. 80-112 (January 1985).

[24] Schilit, B., Adams, N., and Want, R., "Context-Aware Computing Applications", In Proc. of the Workshop on Mobile Computing Systems and Applications (Santa Cruz, CA, Dec. 1994), pp. 85-90.

[25] Asthana, A. and Krzyzanowski, M. C. P., "An Indoor Wireless System for Personalized Shopping Assistance", In Proceedings of IEEE Workshop on Mobile Computing Systems and Applications (Santa Cruz, California, Dec. 1994), pp. 69-74. IEEE Computer Society Press.

[26] Bennett, F., Richardson, T., and Harter, A., "Teleporting - Making Applications Mobile", In Proc. of the IEEE Workshop on Mobile Computing Systems and

Applications (Santa Cruz, California, Dec. 1994), pp. 82-84. IEEE Computer Society Press.

[27] Dey, A., Futakawa, M., Salber, D., and Abowd, G., ”The Conference Assistant: Combining Context-Awareness with Wearable Computing”, In Proc. of the 3rd International Symposium on wearable Computers (ISWC '99) (San Fran_sco, California, Oct. 1999), pp. 21-28. IEEE Computer Society Press.

[28] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri, ”Generic Support for Distributed Applications”, IEEE Computer, pages 68-77, March 2000.

[29] Peter R. Pietzuch and Jean M. Bacon, “Hermes: A Distributed Event-Based Middleware Architecture”, Submitted to the Workshop on Distributed Event-Based Systems (DEBS), 2002.

[30] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks", in Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02). Vienna, Austria, 2002, pp. 639-644.

[31] Antonio Carzaniga and Alexander L. Wolf, “Content-Based Networking: A New Communication Infrastructure”, In NSF Workshop on an Infrastructure for Mobile and Wireless Systems, Scottsdale, AZ, October 2001.

[32] Antony Rowstron and Peter Druschel, “Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems”, In Proc. of Middleware 2001, November 2001.

[33] Tony Ballardie, Paul Francis, and Jon Crowcroft, “Core Based Trees (CBT)”, In ACM SIGCOMM'93, Ithaca, N.Y., USA, 1993.

[34] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel, “Scribe: The Design of A Large-scale Event Notification Infrastructure”, In Proc. of the 3rd Int. Workshop on Networked Group Communication (NGC2001), November 2001.

[35] G. Cugola, E. D. Nitto, and A. Fuggetta, “The JEDI Event-Based Infrastructure and its Applications to the Development of the OPSS WFMS”, IEEE Trans. on Software Engineering, 27(9): 827–850, Sept. 2001.

[36] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D., “Coda: A Highly Available File System for a Distributed Workstation Environment”, IEEE Transactions on Computers 39, 4 (April 1990), 447-459.

[37] Satyanarayanan, M., “Mobile Information Access”, IEEE Personal Communications 3, 1 (February 1996), 26-33.

[38] Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and welch, B., “The Bayou Architecture: Support for Data Sharing among Mobile Users”, In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (Santa Cruz, California, December 1994), pp. 2-7.

[39] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., and Hauser, C., “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”, In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15) (Cooper Mountain, Colorado, August 1995), pp. 172-183.

[40] Mascolo, C., Capra, L., and Emmerich, W., “An XML-based Middleware for Peer-to-Peer Computing”, In Proc. of the International Conference on Peer-to-Peer Computing (P2P2001) (Linkopings, Sweden, August 2001).

[41] Arnold, K., O'Sullivan, B., Scheifler, R. W., Waldo, J., and Wollrath, A., “The Jini Specification”, Second Edition 1999. Addison-Wesley.

[42] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, “Epidemic Algorithms for Replicated Database Maintenance”, Proceedings Sixth Symposium on Principles of Distributed Computing, Vancouver, B.C., Canada, August 1987, pages 1-12.

[43] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz, “An Architecture for a Secure Service Discovery Service”, Fifth Annual International Conference on Mobile Computing and Networks (MobiCom '99), Seattle, WA, August 1999, pp. 24-35.

[44] P. A. Bernstein, “Middleware: A Model for Distributed System services”, Communications of the ACM, vol. 39, pp. 86-98, February 1996.

[45] Linthicum, D., B2B Application Integration: e-Business-Enable Your Enterprise, 2001

[46] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen, “Predicate Matching and Subscription Matching in Publish/Subscribe Systems,” in Proceedings of the Workshop on Distributed Event-based Systems, 22nd International Conference on Distributed Computing Systems, (Vienna, Austria), IEEE Computer Society Press, July 2002.

[47] Szyperski, C., ‘Component Software: Beyond Object-oriented Programming’, Addison-Wesley, 1998.

[48] Kramer, J., and Magge, J., ‘The Evolving Philosophers Problem: Dynamic Change Management’, IEEE Trans. Software Engineering, Vol. 16, Num. 11, pp. 1293-1306. November 1990.

[49] Capra, L., Emmerich, W., and Mascolo, C. 2001b., ‘Reective Middleware Solutions for Context-Aware Applications’, In Proceedings of REFLECTION 2001. The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Kyoto, Japan, Sept. 2001).