

Adaptive middleware for autonomic systems

Simon Dobson, Steve Neely, Paddy Nixon

Publication date

01-01-2006

Published in

Annals of Telecommunications;61(9-10): Pp. 1099-1118

Licence

This work is made available under the CC BY-NC-SA 1.0 licence and should only be used in accordance with that licence. For more information on the specific terms, consult the repository record for this item.

Document Version
1

Citation for this work (HarvardUL)

Dobson, S., Neely, S.and Nixon, P. (2006) 'Adaptive middleware for autonomic systems', available: https://hdl.handle.net/10344/1788 [accessed 25 Jul 2022].

This work was downloaded from the University of Limerick research repository.

For more information on this work, the University of Limerick research repository or to report an issue, you can contact the repository administrators at ir@ul.ie. If you feel that this work breaches copyright, please provide details and we will remove access to the work immediately while we investigate your claim.

Adaptive Middleware For Autonomic Systems

Steve Neely, Simon Dobson and Paddy Nixon

Systems Research Group, School of Computer Science and Informatics,
UCD Dublin, Ireland
steve.neely@ucd.ie

ABSTRACT

The increasingly dynamic nature of resource discovery and binding in modern large-scale distributed and mobile systems poses significant challenges for existing middleware platforms. Future platforms must provide strong support for adaptive behaviour in order both to maintain and optimise services in the face of changing context. We use a survey of existing middleware systems to develop some core themes that characterise and constrain the ability of these approaches to support the development of adaptive and autonomic systems, and draw some possible trends for developing future platforms more appropriate to these domains.

La nature toujours plus dynamique de la decouverte et attachement de services en ce qui concerne les larges systemes modernes mobiles et distribues pose des defis significatifs pour les plate-formes logiciels intermediaires. Ces futures plates-formes devront fournir un large support en ce qui concerne le comportement adaptif afin de maintenir et optimiser les services face a un changement de contexte. Nous exploitons une etude des systemes logiciels intermediaires pour developper des themes centraux qui caracterisent et limitent la portée de ces approches dans le developpement de systemes autonomes et adaptifs, et nous exposons des tendances plausibles pour le developpement de futures plates-formes plus appropriees a ces domaines.

I. INTRODUCTION

Traditional middleware platforms arose as a response to the complexity of constructing multi-vendor enterprise systems. A typical application is to integrate a database, back-office, front-office and point-of-sale systems using a commonly-agreed object model or messaging framework. Most such frameworks focus on providing the tools and techniques to allow developers to perform integration across diverse products and platforms while maintaining the integrity of the overall solution in terms of robustness and reliability.

Recent trends in mobile systems, component-based frameworks, pervasive and autonomic computing have significantly increased the level of dynamism found in systems architectures. Most mobile systems will by definition “encounter” new resources over their lifetime; supporting a component view allows different services for fulfil a given system rôle at different times; while pervasive and autonomic computing seek to maintain the provision of services despite changes in their users' environments or system

characteristics, and indeed often aim to optimise some facet of service delivery against some aspect(s) of this evolving context.

The result of these trends is to intensify the need for middleware integration whilst introducing a range of new challenges not encountered in traditional enterprise systems. Specifically, middleware platforms need to be able to adapt more or less autonomously to changing contexts and situations of use, rather than having their behaviours fixed by human-mediated decisions at design- or configuration-time. Without such adaptation, applications and systems built on the middleware platforms will not be able to exhibit the levels of service demanded of them.

This raises three important questions:

1. To what extent are current middleware platforms suitable for building this next generation of adaptive systems?
2. What deficiencies exist in their support for adaptation?
3. What approaches are emerging to deal with these deficiencies?

In this paper we seek to address these questions by surveying the most important current classes of middleware together with some illustrative systems. We examine each class of middleware with respect to its support for adaptation to changing circumstances, and develop some core themes that both characterise and constrain the ability of middleware to deliver dynamically adaptive behaviour. Finally, we bring together some emerging trends that will shape the development of future middleware systems.

II. THE DEFINING CHARACTERISTICS OF MIDDLEWARE

It is common to think of middleware as synonymous with support for communications between distributed processes across devices. Despite being the main use of the term today, this definition is slightly misleading: the term *middleware* was used in the computing industry before networks of machines were realised. Perhaps the earliest use of the term in the literature can be found in the report of the 1968 NATO Software Engineering Conference [1]: during a discussion on the nature of software engineering D'Agapeyeff described a layer between application programs and service routines, performing operations such as managing a file system, which he identified as *middleware*. D'Agapeyeff's argument was that generic file system functionality was either inefficient or inappropriate for all instances of applications.

In this context middleware is defined as *a layer between application software and system software*. It abstracts the complexities of the system, allowing the application developer to focus all efforts on the task to be solved without the distraction of orthogonal concerns at the system level. For example, the builder of hospital process software does not want to think about initial message analysis and real-time schedulers but patient records, health monitoring equipment, staff rotation schedules and so forth.

With the dawning of the networked era, middleware tools have expanded to address the added complexity encountered in networked applications. In the network-aware system

middleware can be described as *software layered between applications, the operating system and the network communications layers, the purpose of which is to facilitate and coordinate some aspect of cooperative processing*. The middleware provides a *virtual platform* spanning the nodes in the system, and providing a set of communication and co-ordination services that behave (to some extent) as a single system (Figure 1). Different approaches make different concerns transparent to programmers [2], and these variations cause major differences in terms of scalability, expressiveness and adaptability.

We now think of middleware as the "plumbing" within information systems. It routes data and information transparently between different back-end data sources and end-user applications. The common theme throughout all the definitions of middleware is that it is *software that mediates* between varying hardware and software platforms on a network affording group functionality. The core function of middleware is to hide complexity by automating much of the repetitive work common in enterprise systems. In complex distributed environments with arrays of hardware platforms, where intermittent failure is expected and security is paramount the cost of manual integrating can spiral beyond reasonable limits.

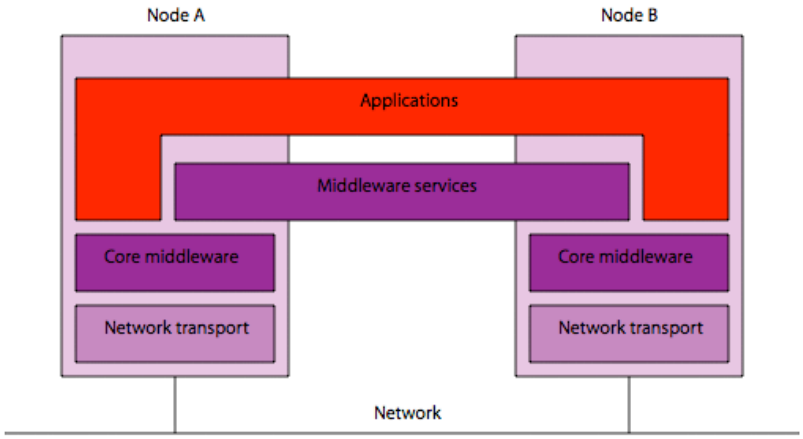


Figure 1 The generic middleware architecture
Architecture generique de logiciels intermediares

Middleware must typically provide support for *naming*, *service location*, *service discovery*, *transport* and *binding*. Services within the system must be redeemable through some symbolic name, allowing the other entities in the system to get a handle to those services to request operations. Service location and discovery mechanisms provide decoupling between entities in the system, dispensing with inflexible hard-coded references to services. A transport mechanism provides a well-defined protocol and data format for exchanging information between network nodes regardless of the operating system and language details of each node. A binding scheme must exist to allow a linking from locally-executing code to some external code. These bindings may take the form of object narrowing in systems such as CORBA, being supplied with an event queue in systems such as IBM's MQ-Series, or some other mechanism.

As well as these basic communication and location services, middleware systems will typically support *replication*, *stable storage*, *concurrency control* and *failure handling* for machine and communication faults. They may transparently deal with these issues, or may provide the programmer with an abstraction above the low-level details. Some systems, for example, provide transactional schemes that will automatically either schedule commit procedures to execute once the system has reached a stable state or run rollback procedures if a failure occurs at any point in the (possibly nested) transaction structure.

Different middleware platforms place different emphasis on security and authentication, ranging from assumptions of trustworthiness (for systems targeting a single LAN), through static rule- and policy-based systems, up to open trust management infrastructures intended for the open mobile internet. Placing such features into the middleware provides common guarantees across a system, but perhaps at the cost of unacceptable performance overheads.

Communications mechanisms of middleware features can vary from request/reply to asynchronous messaging: some implementations provide both. The systems can be language-specific or language-independent: Microsoft's .NET framework is mostly language-independent yet platform-specific, CORBA and most peer-to-peer systems are both language- and platform-neutral.

III. EXISTING MIDDLEWARE STYLES AND ARCHITECTURES

III.1. Object broker systems

Object-oriented programming has become the dominant approach to constructing distributed systems, as with many other fields. Unlike traditional (single-address-space) object-oriented programs, distributed object systems must answer fundamental questions about the impact that distribution has on semantics: if a call is made from a node "a" to a node "b", including as an argument another object on node "a", is the argument copied to the remote node?, or is a remote reference generated to refer back to the original object? How are different sorts of failures managed?, and how are they exposed to programmers? The answers to these questions subtly change the nature of the object system.

A typical object middleware is structured logically as a "container" sitting on each node, which hosts a number of objects. The container provides core management and communications services, typically including object lifecycle management, concurrency control and one or more communications protocols for interaction with objects on other containers. Calls to objects' methods are therefore replaced, at a lower level, by network exchanges. In many cases it is possible for advanced applications to intercept this network traffic and manipulate it in some way, introducing a second mode of interaction into the middleware (call or interception) which can provide great flexibility at the cost of greater potential complexity.

III.1.1. ORBs

Object Request Brokers (ORBs) are perhaps the simplest container architecture, limited to providing a platform on which to build loosely coupled distributed systems with few in-built services. ORBs were envisaged at a time when there was little support for building server applications to fulfil the role of aiding programmers. The early 1990s saw the introduction of the Common Object Request Broker Architecture (CORBA) [3] standard. CORBA was designed to help programmers produce client-server applications as a general purpose integration technology. The commercial success of CORBA made it the *de facto* benchmark for further ORB developments. Most ORBs are implemented as libraries rather than as dedicated container processes.

The initial incarnations of ORBs can be described as Distributed Object Architectures (DOA), which represent an extension of object-oriented programming into middleware. Despite having good internal integration, the observed problem of interoperation between DOA systems lead to the advent of Service Oriented Architectures (SOA), based on open standards such as XML and web services. DOA and SOA are subtly different in terms of the way interfaces are designed, as discussed in [4]: however, in the context of this discussion we can view them as having the same functionality.

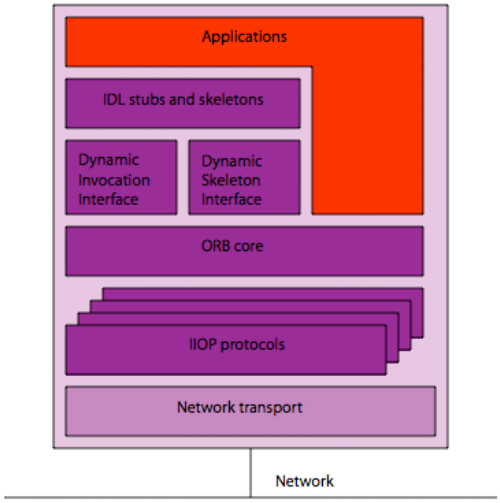


Figure 2 CORBA architecture
Architecture CORBA

The CORBA[1] specifications were drawn up by the Object Management Group (OMG) with the goal of improving interoperability with distributed applications on various platforms. At the core of the middleware platform is the object request broker (ORB), representing that part of the system that handles distribution and heterogeneity issues including all communications between objects and clients. The ORB is typically

[1] For the remainder of this section we will use CORBA as a prototypical example of an ORB. This is to provide a basis the discussion and not intended to suggest that CORBA is the only (or even best) example.

implemented as libraries that the application programmer builds their code against, facilitating the composition of CORBA services.

To enable the calling of services on remote objects their interfaces must be specified. CORBA objects and services are specified using an Interface Definition Language, CORBA IDL, based on a C-style syntax. IDL documents provide the equivalent of a distributed API for accessing published services. To publish a CORBA service a developer publishes the IDL file in some public repository. A client developer passes the IDL file through an interface compiler that outputs "stub" (for client-side) and "skeleton" (for server-side) code in any desired CORBA-compliant language. Before data moves into the ORB the stubs package or *marshal* it into a network-ready format. As data arrives at the other side the skeletons unmarshal it for use by the executing process (Figure 2).
Static definitions via IDL are commonplace in CORBA. However, interest in dynamic CORBA has recently received increased interest due in part to the OMG standardising two scripting languages, CORBAscript [5] and Python [6]. An example of dynamism is the invocation of objects at runtime with the Dynamic Invocation Interface (DII). The DII is a generic invoke operation which takes object reference as input and builds request handling instances on-the-fly. It allows applications to invoke operations on target objects without having compile-time knowledge of their interfaces. This is less efficient than the static stub version as much of the information to execute the object must be provided at runtime. For example the names and types of all arguments and return values must be supplied and dynamically checked.

CORBA applications have three invocation models for method invocation: synchronous, one-way and deferred synchronous. Synchronous requests have at-most-once semantics, in which the caller blocks until a response is returned or an exception is raised. This means that an application must either wait inactive for the duration of the remote call, or must execute remote operations in their own local threads and then integrate the results. One-way requests employ a best effort delivery scheme in which the caller continues immediately without waiting for any response from the server. This is equivalent to a simple point-to-point event service. Deferred synchronous requests rely on the dynamic invocation interface and also have an at-most-once failure semantics. The called continues immediately and can later block until a response is delivered.

In addition it is possible in most ORBs for applications to define interceptors and attach them to invocations of methods on particular objects. The CORBA interceptor definition is expressed in terms of DII, and so remains fairly well-structured.

III.1.2. CORBA services

The CORBA standard defines a set of services designed to support the integration and interoperation of distributed objects. The CORBA services are defined as a layer residing on top of the ORB, as opposed to being part of the core container specification. They are defined by IDL interfaces providing access to their operations. Their significance is that

they provide a standard extension to the middleware platform, raising the level of abstraction at which developers can work.

The *event service* provides facilities for asynchronous communication through events. It is complimented by the *notification service* which has advanced facilities for event-based asynchronous communication. The notification service adds the ability to transmit events in the form of data structures, event subscription, discovery, quality of service and an optional event type repository. The two services encourage the decoupling of objects in the system.

The *naming*, *property* and *trading* services provide mechanisms for identifying, describing and locating objects. Once named object is registered with a trader service it may move around the infrastructure and still usable through dynamic rebinding. The *security* service is the central mechanism for creating secure channels, authorization, access control, policy protection and trust. It defines a very flexible – but perhaps overly complicated – security model for the overall framework.

III.1.3. Building adaptive systems with objects

The CORBA services provide a good initial example of middleware support for adaptive systems development. A CORBA application must acquire references to remote objects with which to interact. In a simple system these would be hard-wired into each application, or stored in a file – neither of which allows easy adaptation to new providers. Using the services, one might (for example) use the naming service to look up the system's trading service, and then use this to acquire other objects. This provides a scheme for managing object acquisition: once an application has bound to the naming service (which may be hard-coded or acquired *via* broadcast), all other bindings are acquired through a service regime. This is appropriate for systems that change relatively slowly, and has been used to great effect in many large enterprise deployments.

Part of Sun's Java Enterprise Edition (J2EE), Enterprise Java Beans (EJB), provides an alternative object-based view of distributed systems. EJB uses Java objects to represent both the state of the system (entity beans) and on-going interactions with clients (session beans), which may be stateful or stateless. Entity beans provide an object façade onto rows of an underlying relational database, and as such cannot be sub-classed – there can be exactly one implementation of a given entity bean within a single application. Moreover all remote interactions occur through session beans, with the entity beans remaining hidden. This simplifies transaction management but can lead to awkward implementation patterns: if an application wants to expose entity beans it must generate a corresponding value object to carry a copy of the entity bean's state across the network. EJB's model is therefore a hybrid between a true ORB and a more service- or document-oriented architecture. It provides a far richer container model including concurrency, lifetime and persistence services as part of the core definition, and the use of Java objects and dynamic loading means that all beans can be managed by a dedicated container process. The definitions of these services are targeted at a particular class of applications exposing relational database table records (one record per object), and – while they

simplify such applications considerably – can provide a poor basis for other application styles.

However, EJB's container constraints also aid some forms of adaptation by system administrators, although not autonomic adaptation. EJB programmers focus on providing fragments of functionality (state manipulation and business methods), with the platform taking care of low-level tasks. This makes it straightforward to (for example) provide a replicated server for high-volume-transaction applications, or to use a federated database for distribution: the application code is not given facilities to over-commit to such architectural choices.

Microsoft's .NET framework provides extensive support for building distributed applications, although without some key enterprise-level services. The more traditional framework approach are typified by ACE [7], which provides abstract classes for basic functions which are then sub-classed to provide application-specific functions, and by Globe [9], which provides caching and replication services within a rich concurrency and consistency model. X-KLAIM [8] and its variants provide a language-level integration of distributed objects into languages such as Java.

An interesting hybrid is provided by Akamai [10]. Akamai is an internet-scale content management network that allows clients to place large content objects (such a movies, audio files or applications) on a global network of managed servers. The servers monitor request traffic and transparently replicate objects on servers close to the request hot-spots, for example having objects "chase the dawn" to be near to countries waking up. By adapting the location of time-consuming content the system provides more scalable and predictable download times while simultaneously moving long-term connections off the clients' networks. In the limit a company could restrict itself to serving only HTML pages locally (requiring little bandwidth) while off-loading images and other content to Akamai. While the details of the adaptation algorithms used are largely proprietary, the service demonstrates the potential for using traffic metadata to adapt underlying storage provision.

III.2. Tuple-space systems

Tuple spaces can be viewed as an implementation of the associative memory paradigm for parallel/distributed computing based on a repository of tuples that can be accessed concurrently. Linda [12] was the first realisation of this concept, This early work in Linda stimulated the development of modern tuple space systems such as JavaSpaces [13] and Jini [14].

Although no explicit work has been reported on tuple space for autonomic systems, they have developed as a middleware platform in domains with similar characteristics, such as pervasive systems. In particular, Equip [15] is a software platform which supports the development and deployment of distributed interactive systems. One key element of Equip is its shared data service, which combines the ideas from tuple spaces and general event systems. This shared data service can be used as communication infrastructure for

context producers and consumers. It supports both querying of current state and receiving events of certain pattern. It also employs replication to improve performance.

Tuple space principles offer a number of advantages for adaptive systems. Firstly they decouple the producer and consumer of data in both time and space: the producer does not know the location of the consumer, and indeed no consumer may exist at the time the tuple is created. Secondly, the tuple space itself may be distributed and persisted to provide robustness for the evolving system, although efficient distribution and searching over wide areas are problematic.

An interesting tuple-space system that directly addresses issues of mobility is Lime [16]. Lime is a multi-space model in which tuple spaces are associated either with locations or with agents. As an agent moves, its local tuple space is dynamically merged with the tuple space at the associated location. Additional primitives allow tuples to be targeted at the tuple spaces associated with particular agents, or to associate code with the appearance in a tuple space of a tuple matching a particular pattern. In many ways this makes Lime a hybrid of tuples with events.

However, although tuple space principles are appealing they have a number of fundamental problems. Tuple spaces do not have a transactional semantics and providing fault tolerance, and in particular rollback, is inconsistent with tuple semantics. This is particularly evident in autonomic systems as the system may evolve and the producers of data may no longer exist when a rollback is initiated [17]. A second concern lies in the scalability and performance of tuple space systems. Some progress has been made in this as evidenced by some recent commercially offerings [18].

In conclusion, tuple space technologies facilitate the necessary decoupling of components in a system and thus enable evolution of the system. However, the observed shortcomings, and in particular fault tolerance, show that further developments are needed to demonstrate their long term viability.

III.3. Message- and event-oriented middleware

Message- and event-oriented systems share a common approach to co-ordination, providing point-to-point or multicast information exchange between largely independent and decoupled clients. They however exhibit significant differences in application and semantics, which tend to result in distinct technologies.

Message-Oriented Middleware (MOM) arose in the mid 1980s as a client-server architecture that decouples communications using messages. A key property of MOM is that it facilitates message passing over heterogeneous platforms encouraging portability, flexibility and interoperability. MOM systems are typically asynchronous and peer-to-peer. Messages are passed from clients to server and stored in message queues until ready for processing, with a guarantee that the message will arrive at the destination process. Networks of message servers are a natural mechanism for database integration, where messages have financial value and guarantees of delivery are paramount.

In MOM systems the client and sever are only loosely coupled. This makes application integration simpler as there are no tight bindings to handle. The systems provide support for a reliable delivery service by keeping queues in persistent storage. The receiver can easily throttle the communications during periods of heavy load and be confident of finding all requests at a more convenient time. MOM systems support the processing of messages by intermediate message server. This could be actions such as filtering, transforming, and logging.

The MOM programming abstraction is rather unwieldy and low-level (at the same level as packets). Dealing with asynchronous communications means that the programmers must write multi-threaded code: the program flow of execution must send the message, spawn a thread to do with the reply and continue on. In such systems a request/reply style of interaction is more difficult to achieve. This begins to push complexity onto the programmers – exactly what the use of middleware was seeking to avoid. The message formats in MOM systems typically are unknown to the middleware, which means the middleware cannot ensure that messages are coherent or type-correct. The queue abstraction only gives one-to-one communication which limits scalability. More importantly for open systems, MOM systems tend to lead to strong vendor lock-in due to incompatible interfaces.

Despite these disadvantages, MOM systems are popular in financial services and other domains where reliability is paramount. They provide a natural framework for database integration and other forms of highly decoupled service composition, and are sufficiently well-specified to allow reconfiguration and re-targeting as the underlying communications services evolve. For more dynamic adaptive systems, however, MOM provides too little support for modelling or metadata collection to assist automated adaptation.

The notion of a message is typically fairly heavyweight, consisting of all the information needed to perform a business-level operation. Many systems require significantly lighter interactions, for example exchanging state changes in a regular basis. Such systems with small messages are typically referred to as event systems. An event is a small packet of information, typically carrying a "payload" indicating a change in the state of an object or component. Event-based middleware systems typically provide one or more event channels to which events may be sent, with the middleware controlling the storage and distribution of the event independently of the producer.

Event-based systems have three major attributes Figure 3). Firstly, in contrast to MOM systems in which a message targets a specific component, an event system decouples event producers from event consumers. An event channel may deliver events to zero, one or many consumers, the population of which may change over time. A single event may be distributed to several points within the system, and may therefore affect the behaviour of the system in a number of different ways in a single operation.

Secondly, event systems can provide a range of semantic models which may be tailored to the needs of applications, selected on a whole-system or per-channel basis. In terms of ordering, channels may provide various guarantees on event delivery ranging from fully ordered (events are delivered to consumers in the order they entered the queue) through producer-ordered (events from a single producer are delivered in the order they were generated) and causally ordered (an event is always delivered before any other events it gave rise to from other producers [11] to completely unordered delivery; in terms of reliability, they may provide best-effort, at-most-once or exactly-once delivery.

Thirdly, event systems have been shown to be extremely scalable. While naive implementations use a central server to host event queues, more advanced implementations may use multiple servers connected by communication spanning trees to distribute events efficiently. The potential efficiency is increased in systems which provide weaker ordering or delivery guarantees.

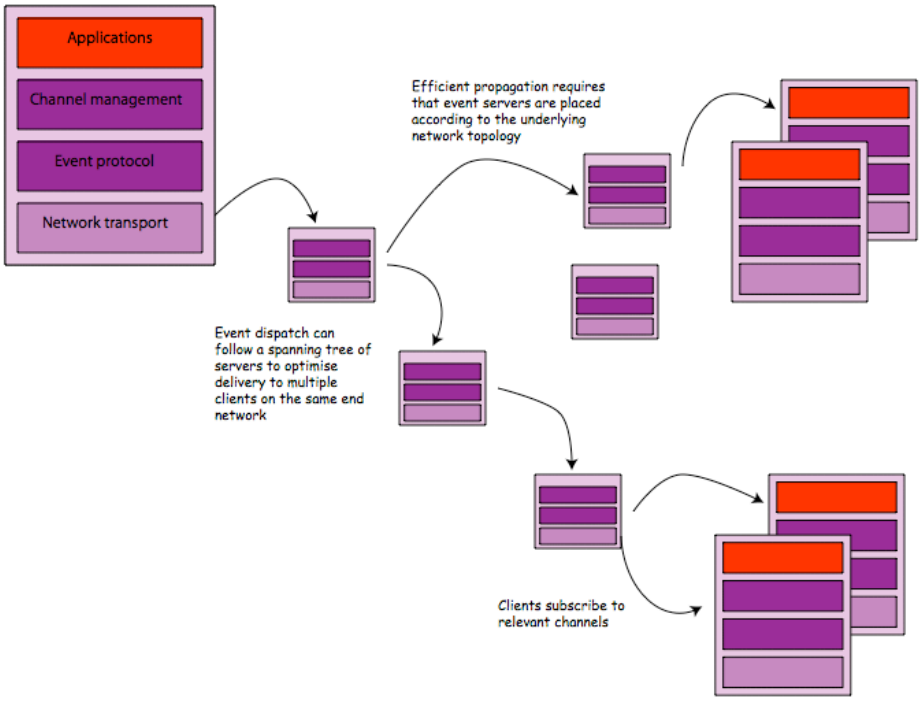


Figure 3 Event delivery architecture
Architecture de livraison d'evenements

Events are typically regarded as asynchronous communications which are delivered to consumers as they become available – sometimes referred to as the push model. An alternative, pull model forces consumers to poll (blocking or non-blocking) for events on a channel. As well as submission and delivery interfaces, event systems must provide a means for specifying the population of producers and consumers. In multi-channel systems, each channel is typically dedicated by the developer, not the middleware) to events of a particular "type". Event channels are then "discoverable" objects which may be acquired by name or by iteration. Systems typically allow both producers and

consumers to subscribe to "interesting" channels. This general architecture is referred to as "publish-and-subscribe" (or "pub-sub") events [19]: consumers subscribe for those events that interest them, and publish them to the appropriate channel.

An enormous variety of systems have been built to populate this landscape. The CORBA event service [20] represents event queues as CORBA objects exposing pull- and push-model interfaces. As with other OMG services, the event service is purely a specification that provides access to underlying implementations (for example [21], [22]). However, the interface specification provides no scope for providing metadata to adapt the event service automatically, which may limit the scope for autonomic management. The Java Messaging Service [23] (JMS) takes a similar approach, closely-integrated with the Java type system and providing client-side filtering based on message attributes.

iQueue [24] takes a more framework-based approach, providing abstract classes to define event structures. It focuses on event filtering and composition, while supporting a range of underlying delivery mechanisms including SOAP and JMS. Siena [25] is similarly framework-based, being perhaps the canonical example of a publish-and-subscribe system whose scalability comes from constructing a spanning tree of servers for event delivery.

One other approach to adaptation in event systems is to overlay the event delivery mechanism onto a lower-level adaptive middleware. The Scribe system [26] leverages the Pastry peer-to-peer infrastructure (see below) to provide adaptive distribution in a fully decentralised fashion. Events channels are represented by keys, with events being retrieved using Pastry's distributed hash table mechanism which provides unordered by highly reliable storage and retrieval. Similarly, Bayeux [27] builds a spanning-tree delivery system on top of Tapestry to support highly efficient delivery over a wide area. The STEAM system [28] acts as a stand-alone middleware using its own MAC-layer protocol to achieve high-volume real-time delivery. Many agent-based systems also use some form of event exchange, coupled with the mobility of individual agents around the network.

Event systems offer two main routes for adaptation. Firstly, the system may allow applications to select different delivery guarantees as appropriate to the semantics of the events being exchanged. This adaptively is typically only available at design time, as different guarantees have different visible effects on applications and so require different design strategies. Secondly, the system may allow the underlying delivery infrastructure to adapt to changing conditions, either autonomously or *via* a management API. User-level management implies the collection of meta-data on service use and underlying network conditions.

While there are obvious similarities between MOM and events, the differences are typically a matter of emphasis. MOM systems focus on point-to-point reliable delivery with predictable ordering, appropriate for database integration and financial exchanges in which these issues are of vital importance. Event systems typically focus on the extent to

which the system can scale to handle high volumes of events being transferred quickly over a wide area.

Although events systems tend to be built bottom-up, exchanging low-level information on system state changes, they can also be used at a higher level. Typically this involves constructing composite events, combining a number of low-level occurrences into single higher-level events. A good example is described by Hayton *et al* [29], who provide a simple algebra for event combination that can be used to "raise the level" of events and allow higher-level composite events to be created from "workflows" of more primitive events. It is an open question whether such techniques would work well in the presence of uncertainty, such as when events are generated from inference [30].

III.4. Peer-to-peer systems

Peer-to-peer (P2P) systems are an area of significant current interest. By removing the need for infrastructural support, P2P systems can potentially support wireless (and other) *ad hoc* networks extremely well, while distributing the load and costs of service provision over the node population. However, this flexibility comes at the cost of reduced reliability and other guarantees. Moreover P2P can offer no guarantee as to whether particular services will be available at all, as all services are provided by the nodes themselves rather than by a managed central provider.

At the protocol level, P2P systems have radically different requirements to the “standard” internet. Supporting these requirements may be accomplished in two ways: by providing a novel transport (MAC layer) protocol to replace IP, or by overlaying a higher-level protocol on top of standard protocols such as TCP/IP. The former has been explored extensively in the context of *ad hoc* wireless networks where there is no legacy infrastructure; the latter is more popular for wired systems, and is essentially unavoidable for systems that seek to span enterprise or the global internet. In either case the P2P network must provide at least (logical) routing, and generally other higher-level services such as node discovery and content management, to applications.

The primary issue for a P2P middleware is routing. In overlay networks, the routing system must translate P2P-level node identifiers into addresses valid for the underlying network (IP addresses for example). There is typically no essential relationship between either the address spaces or the network topology of the underlying and overlayed network.

The Pastry [31] system provides a good example of an overlay P2P system. Nodes in the network are each assigned a 128-bit identifier by a central authority. The identifier space forms a ring topology, independent of the underlying IP addresses of the nodes -- nodes that are "close together" in identifier space may be physically far apart, and *vice versa*. The address space is sparse, so that only a few of the possible addresses in the ring will be alive at any time.

To route packets, each node maintains a routing table mapping nodes that have identifiers close to its own to their IP addresses, learning new nodes by observing the history of

packets it routes. Each incoming packet is routed to a node that the receiver knows about that has an identifier "closer" to the target than the receiver. This strategy is robust against quite substantial amounts of node failure [32], and packets which cannot be routed to their target will end up at a node that is "close" in identifier space (Figure 4)

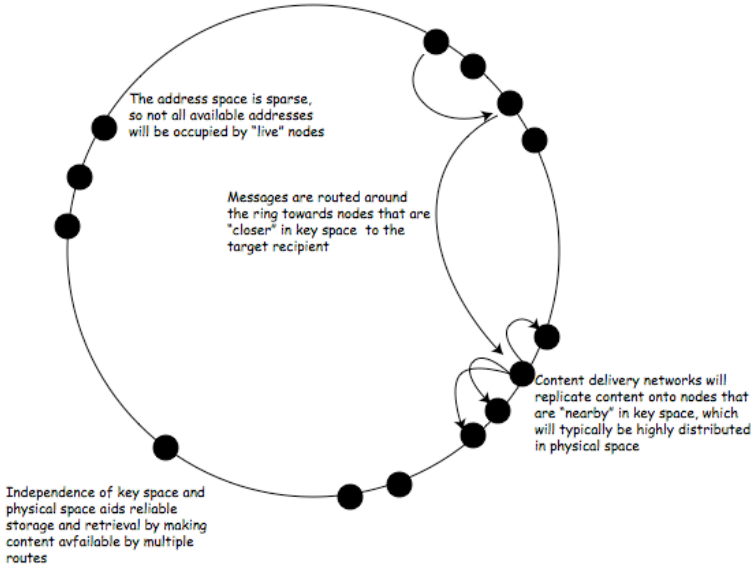


Figure 4 Typical peer-to-peer ring topology for node identifiers
Topologie typique d'un anneau P2P pour l'identification nodale

The main service on top of Pastry is a distributed hash table providing persistent storage. To inject content into the network, a node hashes the content, maps this hash to a node identifier, and routes the content to that node. Routing will take the content either to the target node or "close" to it, with the final node replicating the content onto "nearby" nodes. Retrieval works by querying the node identified by the content's hash. If that node is present, it will have the content; if not, the request will anyway be routed "close" to the node, and this "close" node (or nodes "close" in turn) may have a replica that can satisfy the request. One may adapt this scheme to deal with different predicted frequencies of node dynamism, trading space (more replicas) for robustness (more nodes may leave or fail). It is important to realise that this scheme inherently distributed content across the globe (if run on the open internet) through the independence of node identifiers from IP addresses: even "close" nodes may be physically remote.

Sun's JXTA technology [33] provides a somewhat more structured approach to P2P systems that provides peer groups and service advertisement in the core protocol. Tapestry [34] provides similar content replication to Pastry, augmented with facilities for locating the physically closest replica. This addresses one of the key issues with P2P, that the routing algorithm can result in routes that are significantly (and often perversely) longer than required.

The Intentional Naming System (INS) [35] uses P2P technology to build an infrastructure for locating services using descriptions of their "intention" (print service, display etc), similar in intent to the CORBA trading service. INS constructs a P2P overlay of name resolvers which perform late binding of services against service descriptions. Service requests take the form of data payloads piggy-backed onto service descriptions, allowing service invocation in a single routing step. A service announcement mechanism allows clients and resolvers to discover nodes offering services with particular intentions.

P2P systems are not constrained to use ring topologies: some peer-to-peer systems have topologies targeting specific applications, and some systems (notably T-Man [36]) allow the overlay topology to be adapted dynamically to *any* desired form in a surprisingly small number of local steps without global co-ordination.

P2P systems may also dispense with point-to-point message routing in favour of "gossipped" interactions in which collections of nodes periodically synchronise their local states. Construct [37] uses this approach within the framework of exchanging RDF-structured content. The advantage of this scheme is that queries can be answered using local state: the state may not be completely up-to-date, but its timeliness can be bounded statistically by the properties of the gossiping protocol. This also improves redundancy and reduces communication hot-spots.

IV. FUTURE DIRECTIONS FOR ADAPTIVE MIDDLEWARE

Design and characterisation of middleware platforms for the future is an ongoing active area of research. We have described a view of five main categories of existing systems – ORB-based, message-oriented, tuple-based, event-based and peer-to-peer – highlighting their strengths and weaknesses. In this section we outline the emerging issues and challenges for middleware architects. We indicate areas of work, exhibiting desirable features, worth further application by investigators providing solutions to these problems.

A distillation of the principles and platforms reviewed leads to the following high level research questions:

- **Predictability and stability:** How do we model the system and its properties, monitor these properties and act to maintain stability of the system over time.
- **Fault tolerance and reliability:** Autonomic behaviour fundamentally changes the notion of fault tolerance. We need to understand the fundamental trade-off between adaptation and reliability, and in doing so develop new models of fault tolerance in autonomic systems.
- **Naming and binding:** Naming resources and locating data in a system that evolves requires new approaches to the fundamental systems problem of binding.
- **Programming:** Finally, all the above problems coupled with the evolutionary behaviour of autonomic systems demand new primitives, abstractions and frameworks be provided to the programmer.

Underlying these four core problems below we have identified two key issues: system description and context.

IV.1. Description and meta-description for composition

A core challenge in building middleware systems relates to how we build, share and use understandable descriptions. It should be possible to describe components and their interactions in a way that explicitly prescribes their abstract roles in a system. There must be a mechanism that affords description of not only the information but also the system and its configuration at a high level. We need to be able to reason about the semantics of these descriptions by both manual inspection and automatically. The emergent semantic web technologies are a step towards provision of mechanisms supporting this.

Systems need to be self-describing. Well-designed CORBA enterprise systems are often built in this manner. Components are fully documented, marked-up and registered with a trading (or other) service. Clients in the system can locate, interrogate and activate services based on their metadata descriptions. Understanding of component behaviour and dynamic composition of systems is facilitated in this way.

The essence of middleware is to abstract complexity whilst supporting the combination of distributed services. It should be possible to describe a system as a composition of independent components and connections. Given a *closed set of components* existing systems use registries and traders for this purpose. The known set of components, connectors and behaviour are defined *a priori* by the system designers. These are documented with rules, policies, aspects and so forth, and registered with the appropriate services. Adaptive systems will additionally introduce a variety of composition rules (for the user, applications developer, system, and hardware) which will be generated dynamically. This results in an *open set of components* to be reasoned about. Dynamic composition impacts on the ability to support and maintain robust predictable systems.

This can be illustrated by the following scenario[2]. A user walks through an intelligent building issuing commands to print three documents. The middleware infrastructure in the building adapts to the user movement and routes the request to the nearest printer – with the result that the three documents end up on *different* printers, possibly including devices of which the user in unaware. This example of "print to nearest printer" has appeared several times in the adaptive systems literature: the problem is that the adaptation is *unstable under movement by the user*, and implies that *the space of adaptation is completely known to the user* – neither of which may in fact be the case. Making the "correct" behavioural adaptation requires that the middleware has access to substantial metadata about the space, the users' likely tasks, and so forth.

To realise open adaptive systems, a structure for describing relationships between components coupled with a semantic description of the rules is required. We can use this to describe "closures" of adaptive systems and then analyse whether all the behaviours are "correct". The reverse process allows a given composition to be reused *via* decomposition into constituent parts. It should be possible to reuse components, connectors and architectural patterns even if they have been developed for another purpose. The ability to perform such operation is strongly related to the self-describing

[2] Rather irreverently referred to as the "Dude! Where's my printout?" scenario…

property discussed above. Work on the open standards RDF, RDQL and self-describing formats, such as XML, will play an integral role.

Use of open standards will be a core aspect in the adoption of next generation middleware systems. Traditional middleware was often susceptible vendor lock-in or being closed to the outside world. The CORBA standard introduced IIOP to counter this which has resulted in a highly accessible middleware platform. The move towards openness has manifested itself in the web services platforms with XML, SOAP and the omnipresent set of web technologies.

In many ways the container architecture typical of object-broker systems provides a good platform on which to build adaptive services. The container provides a single point of control within which to manage communication and adaptation, as well as less studied but equally important services such as deployment and update. Such solutions need not be centralised, and – by off-loading complexity from application programmers – may result in more scalable and robust systems with improved potential for commercial roll-out. This implies, however, that the container architecture, semantics and service definitions are defined with proper respect for the autonomic principles they are attempting to support.

A further aspect of openness is to move towards metadata-, rule- and policy-driven adaptation. Metadata ranges from enhanced deployment descriptors to complete metadata models with complex and extensible semantics. This opening of the system architecture – away from the more prescriptive approaches of earlier middleware – has two important effects. Firstly, it makes systems "open-adaptive" [38] by allowing adaptations to be defined post-deployment in response to new techniques or available information. Secondly – and perhaps more importantly – it raises the semantic level at which adaptive behaviour is expressed, making it possible (at least in principle) to reason about the possible effects and interactions of different strategies. This is especially important in systems with minimal human oversight.

IV.2. Trade-offs, context and open problems

It is common to find the notion of optimising *everything* in a system, making everything efficient. There are however many examples of optimisations which are orthogonal: peer-to-peer overlay networks trade-off robustness from random distribution with longer routes to data which slow the systems down; gossiping networks have a massive duplication of data using up memory in trade off to availability. The system must be aware of what is optimising against and trade-off everything else.

Contextualisation of the system and its components will aid in such decisions. Adaptive systems need to be able to contextualise interactions in order to adapt the infrastructure, information or its delivery to the semantics of use. Given the current state, the goals and the context, a more informed decision can be made as to which adaptations to activate. There is a tendency to collect too little metadata on interactions, often in the name of efficiency. This is, however, a false economy: increased instrumentation will enable improved adaptation over the medium term, by providing a larger set of observations

against which to evaluate adaptation decisions. Collection of context needs to be built into the foundations of any adaptive middleware: Construct, for example, treats *all* information sources (including information about its own actions) as sensors integrating into a common model. Without such rich and interconnected context, effective adaptation will be a forlorn dream.

As computational devices become ubiquitous and sensorised, a need for sensor-based middleware emerges. This must be lightweight with an ability to handle a large set of heterogeneous hardware. Open problems with portable devices are new to the middleware developer: for example how does a component rebind to a device that has physically moved off the network? How can such errors be recovered from? How can recovery strategies be expressed without obscuring the main line of the code? These problems will appear in *all* adaptive middleware systems.

Adaptive systems do not involve purely local interactions, and typically need to utilise an appropriate set of local global resources to achieve the task. Furthermore, many local actions have global implications. For example when contacting a (global) key server to authenticate a local node allows communication to begin. Devices must have the ability to move between local environments and retain context, which also requires (at least) non-local information management. Distributed data management technologies such as distributed hash tables and gossiping architectures are starting to address these issues.

Adaptation can interact badly with certain end-to-end properties such as security. The fact that a system is adaptive should not cause it to reveal information that needs to be concealed. Mere observation of a system adapting may allow deductions as to its state by the way it is adapting. A home automation system may give away the absence of the householder by turning the lights off at night. It is unclear whether middleware architectures can address such indirect information flow effectively.

Finally, the latency of adaptation in the system may prove highly problematic. Adaptation requires a system to collect information before arriving at an adaptation decision – but the time required for this data collection may render the adaptation irrelevant. Indeed the same phenomena may be observed in reverse: overly-aggressive (low-latency) adaptation may ignore some of the information needed to make a correct decision. The timeliness of adaptation needs to be considered alongside its correctness, and some of the techniques developed for real-time systems may have a role to play.

V. CONCLUSION

We have presented a survey of the major middleware approaches in current use from the perspective of their support for the development of adaptive and autonomic systems. From this survey we have identified some emerging trends that may be used to shape the development of future middleware platforms that better address these core emerging domains.

All adaptive systems pose fundamental questions about the role of the human administrator: the desire for rapid and flexible adaptation exists in tension with the need

for human intervention in deciding the most appropriate adaptation strategy. It has been suggested[3] that successfully addressing the challenges of autonomic systems may require solving the "strong AI" problem - essentially replacing the human in the loop with an equivalently "intelligent" (in some sense) artificial control system. While no final decision on this conjecture is possible at present (if ever), we believe that trends such as full context modelling, scalable and peer-to-peer resource sharing, and the end-to-end treatment of security and privacy provide mechanisms to substantially simplify the construction, configuration and on-going management of adaptive systems.

It is often forgotten that both adaptation and optimisation are relative terms: one adapts (or optimises) only with respect to some external criteria, with a corresponding impact on other facets of the system. Perhaps the most significant problems for adaptive middleware are (firstly) to allow designers to choose to which parameters must be adapted to, (secondly) to understand which parameters or features must be sacrificed to accomplish this, and (thirdly) to provide the information necessary to make these choices accurately on an on-going basis. These decisions arise externally, from the application domain, rather than internally from the middleware itself, implying the future middleware platforms (of whatever flavour) must embody an increased semantic understanding of the domain in which they reside and their relationship to it. We believe that such well-founded and well-informed adaptive behaviour will the critical enabler for high-performance and high-reliability middleware platforms in the future.

[3] During a panel discussion at the 2nd International Conference of Autonomic Computing 2005, Seattle WA.

REFERENCES

[1] NAUR (P.), RANDELL, (B.) (eds.), Software Engineering, *Report on a conference sponsored by the NATO Science Committee,* Garmisch, Germany, October 1968.

[2] WALDO (J.), WYANT (G.), WOLLRATH, (A.), Kendall, (S.), A Note on Distributed Computing, *Mobile object systems: towards the programmable Internet*, *Springer-Verlag*, pp. 49-64, Jan. 1997.

[3] OBJECT MANAGEMENT GROUP, The Common Object Request Broker: Architecture and Specification, 2.0 ed., July 1995.

[4] BAKER (S.), DOBSON (S.), Comparing Service-Oriented and Distributed Object Architectures, *Proceedings of the International Symposium on Distributed Objects and Applications,* Volume **3760** in *LNCS, Springer Verlag*, Meersman (R.), Tari (Z.), et al (eds), pp. 631-645, 2005.

[5] OBJECT MANAGEMENT GROUP, CORBA Scripting Language, v1.0, *OMG Document formal/01-06-05*, June 2001.

[6] OBJECT MANAGEMENT GROUP, Python Language Mapping Specification, *OMG Document formal/01-02-66*, February 2001.

[7] SCHMIDT (D.), SUDA (T.), An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems, IEE/BCS Distributed Systems Engineering **2** pp. 280--293, December 1994.

[8] DE NICHOLA, (R.), FERRARI (G.), PUGLIESE (R.), KLAIM: A kernel Language for Agents Interaction and Mobility. IEEE Transactions on Software Engineering, **24**(5), pp. 315- 330, 1998.

[9] VAN STEEN (M.), HOMBURG (P.),TANENBAUM (A.S.), Globe: A Wide-Area Distributed System, IEEE Concurrency, **7**(1) (January-March 1999), pp. 70-78.

[10] AKAMAI INC, Home page, "http://www.akamai.com".

[11] ALAGAR (S.), VENKATESAN(S.), Causal Ordering in Distributed Mobile Systems, *IEEE Transactions on Computers*, Volume **46**, Number **3**, pp. 353-361, 1997.

[12] GELERNTER (D.), Generative Communication in Linda, *ACM Trans. Program. Lang. Syst.*, Volume **7,** Number 1, pp. 80-112, 1985.

[13] JAVASPACES PROJECT, Service Specification. http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html

[14] JINI PROJECT, Home Page. http://www.jini.org

[15] GREENBAGH (C.), Equip: A Software Platform for Distributed Interactive Systems, *Technical Report Equator-02-002*, Equator, April 2002.

[16] PICCO (G.P.), MURPHY (A.), ROMAN (G-C), Lime: Linda meets mobility, pp. 368-377 in *Proceedings of the 21st International Conference on Software Engineering*, IEEE Computer Society Press, 1999.

[17] ROWSTRON (A.), Mobile Coordination: Providing Fault Tolerance in Tuple Space Based Coordination Languages, in *Coordination Languages and Models* (Coordination'99), Ciancarini (P.) and Wolf (P.) (eds.), pp. 196-210, *Springer-Verlag*, *LNCS* **1594**, 1999.

[18] GIGASPACES PROJECT, Home Page. http://www.gigaspaces.com/

[19] LIU (Y.), PLALE (B.), Survey of Publish Subscribe Events, *Technical Report TR574*, University of Indiana, 2003.

[20] MOWBRAY (T.), MALVEAU (R.), CORBA Design Patterns, Wiley, 1997.

[21] SCHMIDT (D.), LEVINE (D.), HARRISON (T.), The Design and Performance of a Real-Time CORBA Object Event Service, *Proceedings of OOPSLA'97*, 1997.

[22] O'RYAN (C.), LEVINE (D.), SCHMIDT (D.), NOSEWORTHY (J.R.), Applying a Scalable CORBA Event Service to Large-Scale Distributed Interactive Simulations, *Proceedings of the Fifth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'99)*", 1999.

[23] SUN MICROSYSTEMS, The Java Messaging Service Home Page, http://java.sun.com/products/jms/

[24] IQUEUE PROJECT, Home Page, http://www.research.ibm.com/sync-msg/iQueue.html

[25] CARZANIGA (A.), ROSENBLUM (D.), WOLF (A.), Design and Evaluation of a Wide-Area Event Notification Service, *ACM Transactions on Computer Systems*, Volume **19**, Number **3**, pp 332-383, Aug. 2001.

[26] ROWSTRON (A.), KERMARREC (A.), CASTRO (M.), DRUSCHEL (P.), SCRIBE: The Design of a Large-Scale Event Notification Infrastructure, *Networked Group Communication*, in *LNCS*, *Springer Verlag*, Volume **2233**, pp 30-43, 2001.

[27] ZHUANG (S.), ZHAO (B.), JOSEPH (A.), KATZ (R.) KUBIATOWICZ (J.), Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination, *Proceedings of the ACM Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.

[28] MEIER (R.), CAHILL (V.), STEAM: Event-Based Middleware for Wireless *Ad Hoc* Networks, *Proceedings of the International Workshop on Distributed Event-based Systems*, Vienna, Austria, pp. 585-588, 2002.

[29] HAYTON (R.), BACON (J.), BATES (J.), MOODY (K.), Using Events to Build Large-Scale Distributed Applications, *Proceedings of the 7th ACM SIGOPS European Workshop on Systems Support for Worldwide Applications*, *ACM Press*, pp 9-16

[30] DOBSON (S.), Hybridising Events and Knowledge in an Infrastructure for Context-Adaptive Systems, *Proceedings of the IJCAI 2005 Workshop on AI and Autonomic Communications*, Sterrit (R.), Dobson (S.), Smirnov (M.) (eds.), 2005.

[31] ROWSTRON (A.), DRUSCHEL (P.), Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems, Middleware 2001 in *LNCS*, *Springer Verlag*, Volume 2218, 2001.

[32] CASTRO (M.), DRUSCHEL (P.), KERMARREC (A.), ROWSTRON (A.), One Ring to Rule them all: Service Discover and Binding in Structured Peer-to-Peer Overlay Networks, *SIGOPS European Workshop*, Sept. 2002.

[33] JXTA PROJECT, Home Page, http://www.jxta.org

[34] ZHAO (B.), HUANG (L.), STRIBLING (J.), RHEA (S.), JOSEPH (A.), KUBIATOWICZ (J.), Tapestry: A Resilient Global-Scale Overlay for Service Deployment, *IEEE Journal on Selected Areas in Communications*, Volume **22**, Number **1**, Jan. 2004.

[35] ADJIE-WINOTO (W.), SCHWARTZ (E.), BALAKRSIHNAN (H.), LILLEY (J.), The Design and Implementation of an Intentional Naming System, *Proceedings of the 17th ACM Symposium on Operating System Principles*, 1999.

[36] JELASITY (M.), BABAOGLU (O.), T-Man: Fast Gossip-Based Construction of Large-Scale Overlay Topologies, *UBLCS-2004-7*, Department of Computer Science, University of Bologna, 2004.

[37] STEVENSON (G.), NIXON (P.), DOBSON (S.), Towards a Reliable Wide-Area Infrastructure for Context-Based Self-Management of Communications, *Proceedings of the 2nd IFIP Workshop on Autonomic Communications*, Stavrakakis (I.), Smirnov (M.) (eds.), LNCS, Springer-Verlag, 2005.

[38] ORIEIZY (P.), GORLICK (M.), TAYLOR (R.),HEIMBIGNER (D.)., JOHNSON (G.), MEDVIDOVIC (N.), QUILLIEI (A.), ROSENBLUM (D.), WOLF (A.), An architecture-based approach to self-adaptive software, IEEE Intelligent Systems **14**(3) (May/June 1999), pp. 54-62.