J Internet Serv Appl (2012) 3:95–105
DOI 10.1007/s13174-011-0055-6

SI: FOME - THE FUTURE OF MIDDLEWARE

Another look at the middleware for dependable distributed computing

Mark Little · Santosh Shrivastava · Stuart Wheater

Received: 14 November 2011 / Accepted: 30 November 2011 / Published online: 13 December 2011
© The Brazilian Computer Society 2011

Abstract Key concepts of reliable distributed computing developed during the 1980s and 1990s (e.g., transactions, replication) influenced the standards based middleware such as CORBA and Java EE. This middleware has evolved steadily over the years with message passing facilities to support construction of loosely coupled systems. However, the way networked computing is being used for business and social uses is undergoing rapid changes as new ways of constructing distributed execution environments from varieties of resources, ranging from computational, storage, network to application level services, provided by globally distributed service providers are emerging. In light of these developments, the paper examines what core concepts, components, and techniques that will be required in the next-generation middleware for dependable distributed computing. The paper puts forward the case for five topics for further research: better coordination facilities for loosely coupled systems, restructuring of the middleware stack for supporting multi-tenancy, replication in the large, negotiation, and enforcement of service agreements, and accountability.

M. Little
Red Hat Ltd., Nanotechnology Centre, Herschel Building, Newcastle upon Tyne, NE1 7RU, UK
e-mail: mlittle@redhat.com

S. Shrivastava (✉)
School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU, UK
e-mail: santosh.shrivastava@ncl.ac.uk

S. Wheater
Arjuna Technologies Ltd., Nanotechnology Centre, Herschel Building, Newcastle upon Tyne, NE1 7RU, UK
e-mail: stuart.wheater@arjuna.com

Keywords Middleware · Dependability · Cloud computing · Large-scale systems

1 Introduction

Looking back at the workshops and gatherings held during the last decade or so, on the topic of research directions in distributed computing (e.g., [5]), one can generally single out four major concerns that have underpinned research and development work on middleware: *scalability*, *quality of service*, *manageability,* and *programmability*.

- Scalability concerns the ability to scale in several dimensions; scaling in machine forms—from smart labels to server farms to metacomputing network overlays; scaling in numbers—objects, machines, users, locations; scaling in logical and organisational structures—from ad-hoc collaborations networks to federations of multi-domain enterprises
- Quality of service (QoS) concerns the ability to obtain service-level guarantees such as timeliness, availability, and security. The problem of meeting QoS requirements of applications is made harder in a ubiquitous computing environment where new services and customised services are expected to be added into (existing) applications at an alarming rate.
- Manageability concerns the ability to monitor and control the operation and evolution of large scale, long-lived distributed applications and services, avoiding manual intervention and centralisation (i.e., assumption of one management domain). Distributed applications and services will need to be reconfigured dynamically, for example, to maintain user specified QoS guarantees despite changes in operating conditions (e.g., component failures). Mechanisms are needed to dynamically add,

extend, remove, or move component services in a dependable and predictable manner.
– Programmability concerns the ability to compose new applications from existing applications and services, deploy and maintain them in highly dynamic and heterogeneous computing environments; further, applications are expected to be highly parallel, requiring high-level abstractions necessary for dealing with complexity, with fine-grained resource awareness built according to the end-to-end principle.

Core concepts of middleware for distributed computing (object oriented middleware) were developed during the period between mid-1980s to mid-1990s. CORBA, Java EE, .NET are good examples of industry standards that incorporate many of these concepts. For example, the Arjuna distributed transaction system that the authors were involved in the late 1980s to 1990s [37] is buried inside the widely used JBoss application server middleware, with application developers using Arjunas facilities automatically [29]. Broadly speaking, CORBA/Java EE like middleware enables a client application to invoke operations provided by a remote server application in a type safe manner using remote procedure calls (RPCs). This client-server interaction pattern underpins the abstractions for distributed computing provided by an object request broker (ORB). A designer can group a collection of invocations into an atomic unit by making use of transaction and associated services also attached to an ORB.

Distributed applications are increasingly being constructed by composing them from services provided by various on-line businesses. Service-oriented architectures are seen as providing the framework for building such applications within inter-organisational settings [19]. Although there is much on-going discussion about what constitutes a service-oriented architecture and its relationship to CORBA like architectures (e.g., [3]), there is agreement over the desirability of preserving loose coupling between the partners involved in interactions. CORBA/Java EE like middleware has evolved steadily over the years with message passing facilities (message oriented middleware, MoM), to support construction of loosely coupled systems.

Given this state of affair, it is tempting to think that middleware is more or less done. After all, there is a common set of facilities that implementers need to build their applications: reliable inter-process communication, maintaining distributed state, detecting errors, and recovering from failures and so forth, and by now we ought to have learned what essential tools and techniques we need. The subject of distributed computing has matured with excellent textbooks available (e.g., [25, 39]), the concepts underlying dependability have been expressed precisely [2], and good snapshots of research results on large-scale distributed computing are available (e.g., [24]). However, the way networked computing is being used for business and social uses is undergoing rapid changes. At the time of writing (October 2011), cloud computing is seen as the disruptive technology that is radically expected to change the way companies manage their technology assets and computing needs. The central idea behind cloud computing is that of providing computing resources as services over the network. Cloud services can be at the level of infrastructure (Infrastructure as a service, IaaS), platform (platform as a service, PaaS) where most of the middleware is, and application software (Software as a service, SaaS). It is important to examine in what way this shift in resource provision impacts middleware, which after all is intended to provide the appropriate set of basic abstractions for designing, implementing, and maintaining distributed systems, often in a standards compliant manner.

Before proceeding further, a note of caution: our community is notorious for reinventing distributed computing ideas. Thus, it has been the case that many people working on service-oriented computing (or Grid computing for that matter) have been at pains to differentiate their ideas from existing middleware concepts, often starting from scratch, rather than building on the work of others. A quote from Richard Hamming taken from his 1968 ACM Turing Award lecture is particularly relevant:

> Whereas Newton could say, "If I have seen a little farther than others, it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way.

In our assessment of what core concepts, components, and techniques that will be required in the next-generation middleware for dependable distributed computing, we have come up with five topics for further research: better coordination facilities for loosely coupled systems, restructuring of the middleware stack to enable sharing of resources between multiple tenants, replication in the large, negotiation, and enforcement of service agreements, and accountability.

– *Coordination*: A particularly difficult problem is that of coordinating the interaction between autonomous parties that are in peer-to-peer relationships and loosely coupled (not necessarily on-line at the same time). Business-to-business (B2B) interactions fall into this category. Existing middleware solutions have concentrated on providing client-centric approaches that make use of communication primitives that define semantics of a primitive interaction from a clients perspective (e.g., RPC with

at most once semantics, exactly once message delivery) together with a central coordinator (typically at the client) for coordinating the outcome of distributed application actions into an atomic unit (atomic transaction) or a non-atomic unit (business transaction). The OASIS WS-TX standard [42] is a representative example. We argue that in the world of peer-to-peer relationships, we need messaging abstractions with bi-lateral (multilateral) termination guarantees, together with distributed mechanisms for coordinating the outcome of distributed application actions.

- *Resource sharing*: The concept of many tenants (consumers) sharing resources is fundamental to cloud computing. It is economical for a provider to pool computing resources to serve multiple consumers, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand [30]. The need for resource sharing must be balanced against the conflicting requirements of preserving isolation between tenant applications (the failure of an application must not adversely affect other running applications) and providing appropriate QoS guarantees promised in service agreements with tenants. We argue that the middleware stack needs to be restructured as it is bloated and not suitable for multi-tenancy.
- *Replication in the large*: Data as well as object replication techniques for availability have been studied extensively in the literature. Replication protocols can be categorised by the type of consistency they maintain between the replicas: informally stated, strong consistency, where all replicas have identical states, and weak consistency, where replica states are (temporarily) allowed to diverge. The main drawbacks of using strong consistency replication is the overhead imposed for ensuring that all replicas have the same state; further, in case of a network partition, only the partition with the majority of the replicas (if any) can remain available, the rest must be treated as unavailable. If replicas need not be kept fully consistent all of the time, then lightweight protocols are possible and at the same time availability need not be sacrificed when network partitions occur. Many newly emerged global scale distributed systems (e.g., for on-line analytical processing OLAP—that need to process billions of events in real time) need to maintain large number of replicas for performance reasons (replicas are placed closer to clients to reduce network latency), well beyond what is required just for availability; such systems are finding it hard, if not impossible, to maintain strong consistency. A fresh rethink on exploring the trade-offs between strong and weak consistency, including algorithms and protocols for replication in the large is required.
- *Negotiation and enforcement of service agreements*: In cloud computing, service providers are expected to offer in a rapid manner, on-demand network access to their services to consumers for a fee. As in any business transaction, consumer (client) access to a service will be underpinned by a contract, that we will refer to here as a Service Agreement (SA). A service agreement needs to be negotiated and agreed between the provider and the client before the latter can use the service. Then on, both the client and the provider will need assurances that service interactions are in accordance with the SA, and any violations are detected and their causes identified. There is thus a need for research on automated support for negotiation and enforcement of service agreements.
- *Accountability*: Most users are uncomfortable with the idea of storing their data and applications on systems they do not control [7]. If something goes wrong (say data is lost, or computation returns erroneous results), how do we determine who caused the problem (the customer or the provider)? In the absence of solid evidence, it would be impossible to settle disputes. In other words, cloud services need to be made accountable. Accountability is fundamental to developing trust in services. All actions and transactions should be ultimately attributable to some user or agent. Accountability brings greater responsibility to the users and the authorities, while at the same time holding services responsible for their functionality and behaviour. We argue for a dependable logging service for recording service interactions to enable investigations of accidents or incidents.

In the rest of the paper, we develop these ideas further, beginning with a brief look at the present day middleware for enterprise computing.

2 Now

Present day enterprise middleware has three or more tiers (N-tiers) that provides a modular way of adjusting to changing requirements over time. Typically, the first tier consists of client applications containing browsers, with the remaining tiers deployed within an enterprise representing the server side; the second tier (Web tier) consists of web servers that receive requests from clients and passes on the requests to specific applications residing in the third tier (middle tier) consists of application servers where the computations implementing the business logic are performed; the fourth tier (database tier) contains databases that maintain persistent data for the applications (see Fig. 1).

Applications in this architecture typically are structured as a set of interrelated components hosted by containers within an application server. Various services required by the applications, such as transaction, persistence, security,

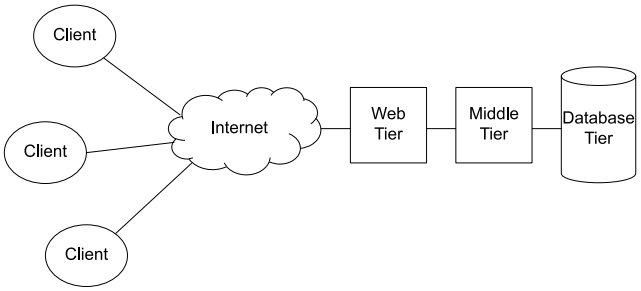

Fig. 1 N-tier architecture

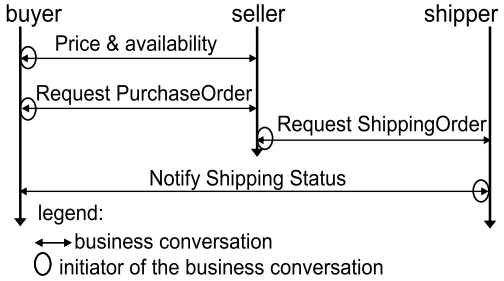

Fig. 2 B2B message interactions

and concurrency control are provided via the containers, and a developer can simply specify the services required by components in a declarative manner. This relieves the developers from the complex task of handling them directly in the components code. At the programming language level, components can be represented as modules, classes, objects, or even sets of related functions. Component technologies have achieved significant progress toward providing composability and interoperability in large-scale application domains.

N-tier architecture permits flexible configuration using clustering within Web and Middle tiers for improved performance and scalability. Availability measures, such as replication, can also be introduced in each tier in an application specific manner. In a typical n-tier system such as illustrated in Fig. 1, the interactions between clients to the web server tier are performed across the Internet. The infrastructure supporting these interactions is generally beyond the direct control of an application service provider that normally only manages the web server tier and the tiers afterward.

This form of middleware provides good facilities for building closely coupled client-server applications. It has been extended with a set of higher level messaging facilities (MoM) with transactions that offer reliable, persistent messaging with well defined operations (e.g., connect, disconnect, deposit message, remove a message) that enable interactions between loosely coupled parties [38]. Java Message Service [20] is an example of a MoM that is part of the Java EE middleware.

3 What next?

3.1 Coordination

We take a closer look at business-to-business (B2B) message-based interactions, that can be seen as two or more interacting parties collaboratively executing a shared activity, such as travel booking, buying/selling of goods and so forth. In such interactions, any peer can initiate the transfer of a message; messages are not necessarily paired as request-response. More importantly, regardless of the message flow, each peer exercises equal control on the status of the activity in the sense that, each peer can locally and unilaterally decide (at any time) on the correctness of a received message and on the final outcome of the interaction.

A primitive B2B interaction (a business conversation) typically involves exchange of one or more electronic business documents for a specific, well-defined function (e.g., verify that a customer credit card is valid and can be used as a form of payment for the amount requested). Industry is developing standards that specify message structures and contents for electronic documents for specific functions, together with a small set of message exchange patterns. RosettaNet partner interface processes (PIPs) [44], ebXML [41] are examples of such standards. A more recent example is the Opentravel Alliance [43] that is standardising on messages for specific travel domains (e.g., Cabin Availability, Cabin Hold, Cabin Un-hold, Create Booking, etc. for cruise holidays).

We consider a simple example to illustrate consistency problems. Figure 2 depicts interactions between three peers (buyer, seller, and shipper) within a B2B application concerned with purchase of goods. The notation used is: a double arrowed line indicates a business conversation; the initiator of a conversation is identified by a circle on the arrow.

The buyer first enquires with the seller about the price and availability of goods. If the required goods are available at acceptable price, the buyer initiates RequestPurchaseOrder conversation with the seller. If this conversation is successful, the seller arranges shipping details with the shipper, who then informs the buyer of shipping details. Such interactions can be viewed as the business partners taking part in the execution of a shared business process (also called public or cross organisational business process), where each partner is responsible for performing their part in the process. Naturally, business process executions at each partner must be coordinated at run-time to ensure that the partners are performing mutually consistent actions (e.g., the shipper is not shipping a product when the corresponding order has been cancelled by the buyer).

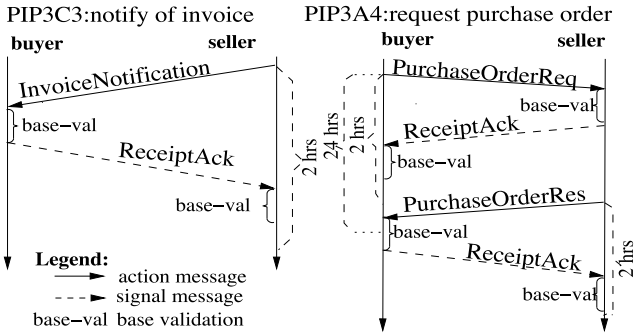

Fig. 3 RosettaNet PIPs

Business conversations are a bit different from the RPC protocols and have several timing and message validity constraints that need to be satisfied for their successful completion. It is worth taking a look at the underlying protocols, so we take a RosettaNet PIP as an example. Each PIP makes use of a specific message exchange pattern made out of business action and business signal (Acks and Nacks) messages. There are two kinds of PIPs: single action and two action (see Fig. 3). In a single action interaction only a single electronic document is exchanged, whereas a two action interaction involves exchange of two documents: a request and its response. A received document is accepted for processing by the receiver only if the document is received within the set timeout period (if applicable) and the document is valid. There are two validity checks that must be met:

- Base-validation: the document must be syntactically valid; this involves verification of a static set of syntactical and data validation rules, according to the specification laid down in the standard; and in addition,
- Content-validation: a base-validated document must also be semantically valid: document contents should satisfy some arbitrary, application specific correctness criteria. This validation varies from trading partner to trading partner and is normally performed within the receivers private business process.

As shown in the figure, the receiver of an action message is obliged to acknowledge it by sending a signal message back within two hours. Since the content-validation is normally performed at the application level (within the private process of the receiver) which could take arbitrary amount of time, the RosettaNet standard specifies that the acknowledgement is sent after base-validation only. Although each PIP performs a conceptually simple action, we face the problem that the PIP initiator (e.g., seller, for PIP 3C3) and its responder (buyer, for PIP 3C3) could end up with conflicting views of a PIP execution. For example, in PIP 3C3, if the ReceiptAck signal message is lost or arrives after the two-hour limit, the buyers and sellers views could respectively be successful and failed termination; subsequent executions of public business processes at each end could diverge, causing business level errors. A conflict can also arise if an action message is delivered (passes base-validation) but not taken up for processing because content-validation fails (so the sender assumes that the message is being processed whereas the receiver has rejected it).

In a loosely coupled system, it could take a long time before such inconsistencies are detected. Subsequent recovery actions—frequently requiring application level compensation—may turn out to be quite costly. For example, assume that the buyer and seller have conflicting views about the outcome of a RequestPurchaseOrder conversation: the buyer regards it as successful (so is expecting goods to be delivered), whereas the seller has failed to complete the conversation successfully, therefore, does not arrange goods delivery. The buyer could wait for a long time (several days perhaps) before suspecting something is amiss. The situation could be even worse if the misunderstanding is the other way round, so the goods are delivered to the buyer who is not expecting them. Existing B2B system architectures do not incorporate any specific solutions for preventing such inconsistencies from appearing at the application level.

Interactions in client-server based systems are based on the client-centric communication primitive, RPC, with well defined semantics (typically at most once) that describes the termination guarantees given to the client in the presence of message loss and server crashes. This enables client programs to incorporate appropriate exception handling when failure exceptions are returned.

In the same manner, in the world of loosely coupled peer-to-peer entities, we need messaging abstractions with bi-lateral (multi-lateral) consistency guarantees. The sender needs a timely assurance that the sent document will be processed by the receiver, and the receiver needs the assurance that if it accepts the document for processing, the sender will be informed of the acceptance in a timely manner; in all other cases, the interaction returns failure exceptions to both the parties. Clearly, a business conversation needs to be encapsulated by some form of synchronisation mechanism to ensure consistent outcome delivery to the participants. Elsewhere, we have discussed what a form such a mechanism could take [32, 33].

An underlying message bus/message broker that provided messaging abstractions hinted above would be an excellent foundation for building higher level facilities for coordinating public business processes. Transactional mechanisms for coordinating long running activities do exist, such as WS-BusinessActivity [42], but so far their take-up has not been encouraging. One reason is the reluctance of industry to accept protocols that require cross-industry coordination (which implies loss of autonomy). It is also the case that in peer-to-peer interaction settings, centralised coordinator based solutions sit awkwardly. For example,

looking at Fig. 2, it is not obvious, who should own the WS-BusinessActivity coordinator and how the ownership issue should be resolved. We need an alternative, a distributed solution that avoids the need for a central coordinator and at the same time has minimum impact on organisational autonomy.

3.2 Resource sharing

At the infrastructure level (IaaS), it is economical for providers to manage server clusters (datacenters) whose machines can be shared between as many tenants as possible. Similar economic arguments apply at the levels of PaaS and SaaS: PaaS level middleware services (such for reliable messaging, transaction management) and SaaS level applications should be shareable between multiple tenants. Recalling the remark made earlier, the need for resource sharing must be balanced against the conflicting requirements of preserving isolation between tenant applications (the failure of an application must not adversely affect other running applications) and providing appropriate QoS guarantees promised in service agreements with tenants. At the level of IaaS, hypervisor based virtualisation technology provides a sound basis for sharing a machine by multiplexing it between a number of virtual machines (VMs). Let us examine the state of affairs at the next level, PaaS. Figure 4 shows a simplified view of the principal number of software systems that would be installed in a VM for running a Java application: guest operating system followed by a Java virtual machine (JVM) followed by an application server whose containers will host the application.

Looking at the system as a whole from hardware up, the first impression, quite rightly, is that of a bloated software stack where similar functionalities are being duplicated in many places. The disadvantages are clear: enormous consumption of storage space, performance takes a hit, and the reliability degrades, simply because of the increased amount of software, and the interaction requirements between them. Security is jeopardised, because of the increased amount of vulnerabilities, and various important non-functional properties are harder to control because they are influenced in so many places in the running middleware.

A single application server is expected to support the execution of multiple applications (as depicted in virtual machine 1). The guest operating system is normally a multiprogramming operating system that comes with a variety of mechanisms for supporting multiple users (protection, resource usage accounting and so forth) and their processes with separate address spaces provide good isolated execution environments. Unfortunately, these mechanisms are not directly available to the application server but to the JVM which was originally introduced for achieving code mobility and not for hosting multiple applications. A JVM typically

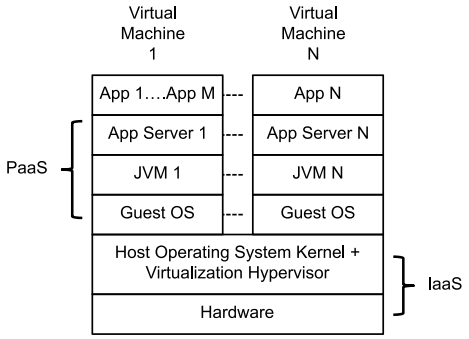

Fig. 4 Middleware stack

hosts—within a single address space the entire application server middleware for running applications. Running multiple applications as threads within a single address space is not desirable due to lack of isolation: a fault in one application can lead to the corruption of other application and JVM objects, leading to crash of the entire process.

A practical consequence of this is that application servers are typically deployed to host a single application (as depicted in virtual machine N). Not only is this is extremely wasteful of storage space, it also makes differentiated resource control for applications difficult. Results from past research efforts to address this problem by introducing multi-programming and resource control capabilities to JVM (e.g., [22]) have not been adopted into the Java middleware, but there are renewed efforts to incorporate multi-tenancy features in the future versions of the Java middleware [21]. The fundamental problem with managed run time environments (such as JVM) is that they end up duplicating the operating system features. An alternative is to do away with them and run compiled languages, an approach adopted in the concurrent programming language Go [13] that nevertheless provides features common to managed run time environments (garbage collection, run-time reflection). However, it looks likely that JVM and the likes are here to stay, so we need better ways of incorporating resource sharing capabilities in multilevel systems without duplicating operating system features.

3.3 Replication in the large

n the N-tier architecture, relational database systems (database tier, Fig. 1) have traditionally provided the persistent storage facility. Stored data is manipulated using serialisable, ACID transactions that keep the data in a strongly consistent state (each transaction sees the latest committed state of the data). When replication is used for availability, the aim is to provide the same strong consistency guarantee (also termed one-copy serialisability).

3.3.1 Consistency and transactions

The design of present day relational database systems is based an architecture consisting of a database server with the storage system of disks attached directly to the server or made accessible via a storage area network. Such a design is not suitable for exploiting massive compute and storage facilities available in datacenters built from commodity machines with directly attached disk storage. This limitation is one of the reasons behind the recent interest in designing petabyte scale distributed storage systems running over datacenters capable of serving a variety of workloads generated by very large number millions of users. Failures are inevitable when large number of machines and disks are involved, so any such design will need to use replication for availability. Earlier we mentioned the tension that exists between maintaining good performance, high availability and strong consistency.

Essentially, under strongly consistent replication, performance takes a hit because write operations need to be performed synchronously (meaning that a write operation completes only after the value has been delivered to all the replicas), whereas under weakly consistent replication, write operations can be performed asynchronously (meaning that a write needs to complete only on a single copy and others can be updated in the background). Further, under network partitions, weakly consistent replicas in all the partitions can remain available, whereas this cannot be permitted under strong consistency protocols, that can only allow at most the majority partition (if any) to remain available. We also note that replicating data across machines within a single datacenter provides tolerance against machine crashes, but tolerance against datacenter breakdown (e.g., due to some regional disaster) is still required and demands that data must be replicated across geographically separated datacenters. The CAP theorem states that all the three properties of consistency, availability, and tolerance to network partitions cannot be achieved simultaneously, but any two can be achieved [12].

It is instructive to see what choices some prominent distributed storage systems deployed within clouds Dynamo [9], Cassandra [26], Megastore [4] used for supporting Google App Engine PaaS and Windows Azure Storage, WAS [6]—have made. A feature common to all of the above (and a host of similar) storage systems is that they are structured simply as key-value stores: data is subdivided (sharded) into application specific units (a blob) identified by unique keys; sophisticated features of relational databases (e.g., joins, SQL queries) are not supported. Dynamo and Cassandra do not provide transactions and strong consistency (a read operation on a data item in a blob is not guaranteed to return the value last written). The CAP theorem is often sited as the reason for favouring availability over consistency in these systems, but the fact is that they do not provide strong consistency even if there are no network partitions; rather consistency is sacrificed primarily to obtain low latency read/write operations (an important goal). Although suitable for specific classes of applications, we claim that weakly consistent storage systems are not a satisfactory basis for supporting general purpose applications, as they substantially complicate the application builders task who needs to deal with the consequences of accessing stale data in the application code.

Megastore and WAS systems suggest a way forward: they have been carefully engineered to support strong consistency (in the absence of network partitions), and provide transactional access to data items within a blob. Transactional access to data within multiple blobs spread across datacenters is not supported. The main reason is the increase in latency this entails, as it requires running replication and two phase commit protocols across geographically separated datacenters.

We suggest three areas of research: (i) A vast majority of applications will continue to rely on relational database systems for their persistent storage needs. Thus, a substantial restructuring of their monolithic design to take advantage of datacenters is required. The appearance of products such as Xeround Cloud Database [45] suggests that this is beginning to happen. (ii) Supporting distributed transactions spanning multiple datacenters: are there any ways of reducing latency here? A recent research paper [28] proposes an architecture that avoids the need for two phase commit across datacenters (but pushes the complexity to a logically centralised transaction manager). (iii) Primary memory caching techniques for disk data are most effective in reducing latency when the workload is read intensive. Distributed caching techniques within a cluster of machines (e.g., Memcached [17], Infinispan [15]) are in wide use. Looking at Fig. 1, caching can be used in all the three tiers, perhaps simultaneously. When data is kept in many caches, it is most likely that keeping them all strongly consistent will prove counterproductive. A fresh rethink on exploring the trade-offs between strong and weak consistency is thus needed.

3.3.2 Responsive group communication

Under group communication we include services for membership management and reliable total order (atomic) multicast to group members: services that are deemed essential for implementing configuration management, load balancing and replication in distributed systems. Megastore and WAS make use of Paxos algorithm based total order services for replica management [27]; JGroups reliable group communication system [16] is used within the JBoss application server cluster for load balancing and managing Infinispan caches [15]. JGroups has also been used for replicating Java

components [23]. Group communication systems, of necessity, need to use multi-round protocols to reach consensus on membership and message ordering. Existing systems do not scale well beyond tens of members. This is a serious limitation, as in a datacenter environment, there are cases when group membership can be in thousands rather than in tens (e.g., a cluster of several thousand machines acting as a distributed Infinispan cache). We put forward responsive group communication, taken here to mean low latency group communication capable of scaling to thousands of members, as a topic for further research. Below we examine the underlying issues that hinder scalability and hint at possible solutions.

Fault-tolerant distributed algorithms that underpin group communication systems have been designed under the assumption of either synchrony (bounds on processing and communication delays exist and are known) or asynchrony (bounds are unknown and it is not a failure when an estimate does not hold). Of necessity, large scale system designs have opted for the latter. However, asynchronous algorithms can only be designed to ensure absolute correctness at the cost of everything else. Their design is quite complex for error-free implementations and overhead high even when delay estimates hold; when estimates do not hold, e.g., when bursts occur, delay, and overhead further increase due to unproductive message exchanges triggered by false timeouts. Is it possible to do better, by developing a model that regards the validity of (tunable) estimates in probabilistic terms? Emerging hosting environments, e.g. datacenters, certainly motivate such a model.

A recent paper [1] critically examines the implications of using Paxos style solutions for building fault-tolerant systems for datacenters and enterprise network based applications. The paper then lays down a convincing rationale for seeking design alternatives to considering asynchronous model augmented with imperfect failure-detectors, and advocates novel ways for building perfect failure detectors. A different strategy is proposed in [10]. Taking a cue from the approach adapted in an earlier system, Total, that made use of LAN broadcast as the transport layer for building total order multicast [31], it proposes using a reliable multi-cast system that offers attainable multi-cast delivery guarantees in probabilistic terms as the transport layer (see [11] for additional details).

3.4 Negotiation and enforcement of service agreements

When services are provided and consumed on a fee paying basis, both the client and the provider will need assurances that service interactions are in accordance with the service agreement (SA), and any violations are detected and their causes identified. Enterprise middleware needs to be enriched with services for automated support for negotiation, monitoring and enforcement of service agreements. We summarise our take on this subject, based on work presented elsewhere [34, 35].

Ideally, it should be possible to encode an SA as a set of executable business policies that can be evaluated by either party for controlling service interactions. Typically, a provider will have a set of local (private) business policies (LP) for customising an SA for different classes of clients. Figure 5a shows a simple scheme where the provider uses a Policy Manager (PM) module (loaded with an executable versions of SA and LP) for controlling access to the service by the client. The gateway acts as a policy enforcement point that either allows or prohibits access to the service as directed by the PM.

In Fig. 5a, the decision whether the clients service access is compliant with respect to the SA is taken by the PM of the provider; however, there may be situations where the clients organisation independently wants to perform such a compliance check, in which case, the symmetric deployment scheme of Fig. 5b is relevant. The clients organisation might have its own local policies that put additional constraints on who/when service access is permitted (e.g., a local policy might be that only a senior manager is permitted access). Another deployment possibility is depicted in Fig. 5c: here an independent third party is responsible for checking SA compliance, whereas the parties only check for their local policy compliance. The configuration depicted in Fig. 5b opens up the possibility of the two PMs being able to interact and negotiate to install a new SA on the fly. For example, a customer of a service might wish to upgrade to become a premier customer, in which case a new SA will come in force. This possibility is hinted at in Fig. 5d where SA is under negotiation.

We expect policy managers and gateways to play increasingly important roles in distributed execution environments constructed from cloud based services. The machine interpretable language used for encoding SA and LP should be expressive and usable. By usability, we mean that a technical person who understands SAs and LPs written in a natural language should be able to translate them into executable versions with relative ease. By expressiveness, we mean that the language should be widely applicable. These are challenging goals to achieve as the intended meaning of clauses expressed in a natural language can be remarkably hard to capture and represent in a rigorous and concise manner for computer processing. Further, we require that the encoded versions of SAs and LPs be amenable to formal analysis, meaning there should be tools available for validating the logical consistency of an SA and LP taken individually and together. Such tools are urgently needed.

3.5 Accountability

Earlier we stated that cloud services need to be made accountable. Accountability brings greater responsibility to

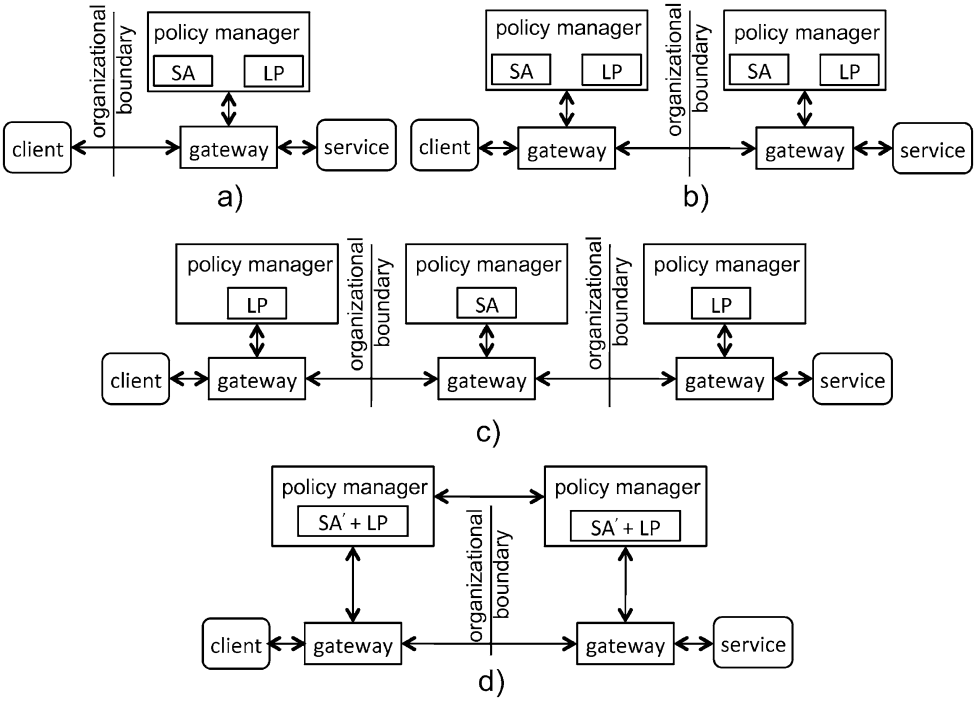

Fig. 5 Policy managers and gateways

the users and the authorities, while at the same time holding services responsible for their functionality and behaviour. We argue for a dependable logging service for recording service interactions to enable investigations of accidents or incidents. Such a logging service should underpin the policy based monitoring and enforcement service discussed in the previous section.

The legal implications of data and applications being held by a third party are complex and not well understood [36]. We take inspiration from the civil aviation industry and the way it has organised itself in providing one of the safest modes of world wide travel. Under internationally agreed regulations, a commercial aircraft must carry a tamper proof black box an event recorder—to record the performance and the condition of the aircraft in flight (actually, the recorder has two components, the cockpit voice recorder for recording radio transmissions and sounds in the cockpit, such as the pilots voice and flight data recorder that monitors parameters such as altitude, airspeed and heading). The classic use of the event recorder is in the investigation of an incident or accident. The event recorder is an impartial and highly reliable witness which can provide a great deal of information to investigators about the circumstances surrounding the event, the actions taken by people involved, and what happened during the event. Event recorders can be used in legal investigations to uncover signs of negligence or improper operation, or to reveal that an accident was genuinely an accident, caused by something like a freak occurrence which disabled the systems on the aircraft.

What we need is a 'flight recorder' for the cloud that records (logs) service interaction events at sufficiently fine granularity in a tamper proof manner to enable investigations of accidents or incidents. Logging should have the following characteristics:

- Logging must guarantee fairness and non-repudiation, ensuring that well-behaved parties are not disadvantaged by the misbehaviour of others and that no party can subsequently deny their participation.
- It should enable tracing back the causes of an 'incident' (behaviour that is a deviation from expected behaviour) after it has occurred.
- Its presence must not limit the functioning and the types of services offered by the cloud itself.
- Logging itself should be constructed using cloud computing services.

These are tough research challenges. Ideally, the middleware should give us trusted containers that provide a secure execution environment for the deployment of components in general and logging in particular. The use of Trusted Platform Module (TPM) hardware facility now available in commodity processors [18] that are tamper-evident under hardware attack and tamper-resistant to software attacks, along the lines suggested in [14] together with middleware based approaches for non-repudiable interactions [8, 40] suggests a way forward.

4 Concluding remarks

The authors have fond memories of participating in IEEE workshops on configurable distributed systems (held in the

mid-1990s, now sadly discontinued), where we used to discuss how to make changes to distributed systems on the fly. This was pretty speculative work, as there were not that many distributed systems around, let alone the need for dynamically reconfiguring them. Matters have changed dramatically since then as distributed systems have arrived in a big way. In our assessment of what core concepts, components, and techniques that will be required in the next-generation middleware for dependable distributed computing, we have taken two related factors into account: (i) distributed applications are increasingly being constructed by composing them from services provided by various online businesses; and (ii) increasing use of computing services that are provisioned using cloud computing. These two factors have influenced our choice of topics for further research: better coordination facilities for loosely coupled systems, restructuring of the middleware stack to enable sharing of resources between multiple tenants, replication in the large, negotiation and enforcement of service agreements, and accountability. For each topic, we have discussed the underlying issues and suggested possible directions of research.

References

1. Aguilera M, Walfish M (2009) No time for asynchrony. In: Proc USENIX hot topics in operating systems (HotOS09), Berkeley, USA
2. Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33
3. Baker S, Dobson S (2005) Comparing service-oriented and distributed object architectures. In: Meersman R, Tari Z et al (eds) Proceedings of the international symposium on distributed objects and applications. Lecture notes in computer science, vol 3761. Springer, Berlin
4. Baker J et al (2011) Megastore: providing scalable, highly available storage for interactive services. In: Conference on innovative data systems research (CIDR 11). Asilomar, California, USA
5. Schiper A et al (eds) (2002) International workshop on future directions in distributed computing, Bertinoro. Lecture notes in computer science, vol 2584. Springer, Berlin
6. Calder B et al (2011) Windows Azure Storage: A highly available cloud storage service with strong consistency. In: 23rd ACM symposium on operating systems principles (SOSP). Cascais, Portugal
7. Chow R et al (2009) Controlling data in the cloud: outsourcing computation without outsourcing control. In: ACM cloud computing security workshop, CCSW09, Chicago, Illinois, USA
8. Cook N, Robinson P, Shrivastava S (2004) Component middleware to support non-repudiable service interactions. In: IEEE/IFIP international conference on dependable systems and networks (DSN 2004), Florence, pp 605–614
9. DeCandia G et al (2007) Dynamo: Amazon's highly available key-value store. In: 21st ACM symposium on operating systems principles (SOSP), New York, NY, USA, pp 205–220
10. Ezhilchelvan P, Shrivastava S (2010) Learning from the past for resolving dilemmas of asynchrony. In: 3rd ACM SIGOPS international workshop on large scale distributed systems and middleware, SIGOPS. Oper Syst Rev 44(2)
11. Ezhilchelvan P, Clarke D, Di Ferdinando A (2011) Near certain multicast delivery guarantees amidst perturbations in computer clusters. Technical Report No CS-TR-1267. School of Computing Science, Newcastle University
12. Gilbert S, Lynch N (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM SIGACT News 33(2)
13. golang.org
14. Haeberlen A (2010) A case for the accountable cloud. In: 3rd ACM SIGOPS international workshop on large scale distributed systems and middleware. SIGOPS Oper Syst Rev 44(2)
15. http://www.jboss.org/infinispan
16. http://www.jgroups.org
17. http://memcached.org/
18. http://www.trustedcomputinggroup.org/
19. Issarny V et al (2011) Service-oriented middleware for the Future Internet: state of the art and research directions. J Internet Serv Appl 2(1):23–45
20. Java Message Service Specification www.oracle.com/technetwork/java/index-jsp-142945.html
21. Java Specification Request JSR 342: Java Platform, Enterprise Edition 7 (Java EE 7) specification. http://jcp.org/en/jsr/detail?id=342
22. Jordan M et al. (2004) Scaling J2EE application servers with the multi-tasking virtual machine. Sun Microsystem Technical Report, SMLI TR-2004-135. Available at: https://labs.oracle.com/techrep/2004/smli_tr-2004-135.pdf
23. Kistijantoro A, Morgan G, Shrivastava S, Little M (2008) Enhancing an application server to support available components. IEEE Trans Softw Eng 34(4):531–545
24. Krakowiak S, Shrivastava S (eds) (2000) Advances in distributed systems. Lecture notes in computer science, vol 1752. Springer, Berlin
25. Kshemkalyani AD, Singhal M (2011) Distributed computing: principles, algorithms, and systems. Cambridge University Press, Cambridge. ISBN: 9780521189842
26. Lakshman A, Malik P (2010) Cassandra a decentralized structured storage system. In: 3rd ACM SIGOPS international workshop on large scale distributed systems and middleware. SIGOPS Oper Syst Rev 44(2)
27. Lamport L (1998) The part-time parliament. ACM Trans Comput Syst 16(2):133–169
28. Levandoski J, Lomet D, Mokbel M, Zhao KK (2011) Deuteronomy: transaction support for cloud data. In: Conference on innovative data systems research (CIDR 11), Asilomar, California, USA
29. Little M, Shrivastava S (2011) The evolution of the Arjuna transaction processing system, dependable and historic computing. In: Jones CB, Lloyd JL (eds) Lecture notes in computer science, vol 6875. Springer, Berlin, pp 323–343
30. Mell P, Grance T (2011) The NIST definition of cloud computing. NIST Spec Publ 800-145
31. Melliar-Smith PM, Moser L, Agarwala V (1990) Broadcast protocols for distributed systems. IEEE Trans Parallel Distrib Syst 1(1):17–25
32. Molina-Jimenez C, Shrivastava S (2006) Maintaining consistency between loosely coupled services in the presence of timing constraints and validation errors. In: 4th IEEE European conference on web services (ECOWS), Zurich, pp 148–157
33. Molina-Jimenez C, Shrivastava S, Cook N (2007) Implementing business conversations with consistency guarantees using message-oriented middleware. In: 11th IEEE international EDOC conference (EDOC 2007), Annapolis, Maryland, USA, pp 51–62
34. Molina-Jimenez C, Shrivastava S, Strano M (2009) A model for checking contractual compliance of business interactions. IEEE Trans Serv Comput. doi:10.1109/TSC.2011.37

35. Molina-Jimenez C, Shrivastava S, Wheater S (2011) An architecture for negotiation and enforcement of resource usage policies. In: The IEEE international conference on service oriented computing applications (SOCA 2011), Irvine, CA, USA
36. Mowbray M (2009) The fog over the Grimpen Mire: Cloud computing and the law. SCRIPTed J Law Technol Soc 6(1)
37. Parrington G, Shrivastava S, Wheater S, Little M (1995) The design and implementation of Arjuna. Comput Syst 8(3):255–308
38. Tai S, Rouvellou I (2009) Strategies for integrating messaging and distributed object transactions. In: Sventek J, Coulson G (eds) Middleware 2000. Lecture notes in computer science, vol 1795. Springer, Berlin, pp 308–330
39. Tanenbaum AS, van Steen M (2007) Distributed systems: principles and paradigms. Pearson, Upper Saddle River. ISBN: 0132392275
40. Wang C, Chen S, Zic J (2009) A contract-based accountability service model. In: Proc of IEEE international conference on Web services (ICWS 2009), Los Angeles, CA, USA
41. www.ebxml.org/specs
42. www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx
43. www.opentravel.org
44. www.rosettanet.org
45. xeround.com