Gossip-based Service Coordination for Scalability and Resilience

Filipe Campos
Qimonda Portugal S.A.
(Trainee/Internship, 2008)
fcampos@di.uminho.pt

José Pereira
Universidade do Minho
jop@di.uminho.pt

ABSTRACT

Many interesting emerging applications involve the coordination of a large number of service instances, for instance, as targets for dissemination or sources in information gathering. These applications raise hard architectural, scalability, and resilience issues that are not suitably addressed by centralized or monolithic coordination solutions.

In this paper we propose a lightweight approach to service coordination aimed at such application scenarios. It is based on gossiping and thus potentially fully decentralized, requiring that each participant is concerned only with a small number of peers. Although being obviously simple and scalable, it has been shown that gossip-based protocols lead to emergent strong resilience guarantees.

We illustrate the approach with WS–PushGossip, a proof-of-concept coordination protocol based upon the WS–Coordination framework. Besides presenting WS–PushGossip, we illustrate its usefulness with a sample application, and outline a middleware implementation based on Apache Axis2.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.2.11 [**Software Architectures**]: Patterns

General Terms

Design, Performance, Reliability

Keywords

Web Services, Gossip

1. INTRODUCTION

As service-oriented computing matures and becomes widespread, there is an increasing demand for applications involving very large numbers of coordinated services. For instance, in systems management it is often necessary to aggregate and then query information amassed from a large number of sources. More often, the goal is to disseminate information to a very large number of interested parties, as attested by the growing interest in notification services, as described in Section 2.1.

As an example, consider a trading floor scenario in which stock market information is disseminated to a number of trader workstations and automatic trading systems. This way, each node maintains a local copy of the list of stock values with which a client application may interact.

This scenario has traditionally been addressed by monolithic applications and group communication protocols [25], but it is increasingly interesting in a service-oriented approach as stock markets and trading systems become increasingly interconnected and interoperable. Anecdotal evidence for this is its usage to motivate multiple research efforts [23, 15, 14] and also as sample code for popular middleware packages [1].

Stock trading systems have however very stringent resilience and scalability requirements, that are hard to achieve even with existing monolithic implementations [25]. Specifically, it is very hard to achieve stable high throughput when the number of participants is very large, even if the network topology and conditions are stable. Such stability is an essential guarantee for these systems where high volumes of data are transferred with tight timeliness requirements. The same requirements exist, for instance, in automated production management systems as deployed in the semi-conductor industry.

Furthermore, it has been pointed out that this is a fundamental limitation of reliable information dissemination based on feedback mechanisms [11]. The problem stems from messages being buffered at multiple locations until fully acknowledged by all destinations, to deal with node and network faults. A single slow receiver, or worse yet, multiple transient perturbations, can thus delay acknowledgment and garbage collection, leading to degraded throughput.

Current state-of-the-art is that stable high throughput can be achieved by using gossip-based, or epidemic, protocols [12]. As described in Section 2.2, such protocols are also highly resilient to network and process faults, while scaling to large number of participants and high message throughput. Gossip protocols are, for instance, a key technology within Amazon.com Web Services implementation infrastructure [28].

The goal of this paper is to leverage gossiping in service-oriented computing as an high level structuring paradigm, thus inherently achieving scalability and resilience when coordinating large numbers of services.

MW4SOC '08, December 1, 2008, Leuven, Belgium

In detail, we aim at using gossip regardless of the system being architected according to existing event dissemination and notification standards, and with minimal to none application code changes. We illustrate that this is feasible with the WS–PushGossip proof-of-concept service based on the WS–Coordination framework.

Note that being just a proof-of-concept, WS–PushGossip is restricted to a single gossip style, which is useful only for information dissemination, and does not consider security issues. Future work should expand its applicability and improve performance to match current state-of-the-art in gossip-based protocols and fully integrate with other Web Services standards.

The rest of the paper is structured as follows. We start by presenting brief surveys of the state of the art in information dissemination services and gossiping protocols in Section 2. Section 3 describes the architecture of the proposed WS–PushGossip service, and Section 4 its implementation. Finally, Section 5 discusses the approach and future work directions.

2. BACKGROUND

In this section we survey the state of the art in notification services and gossip-based dissemination. Gossip is also useful for *aggregation* of information located at disparate nodes in a network [19, 26], specially when the targeted network has a large number of nodes which provide some useful data to be conveyed. This is however out of the scope of this paper.

2.1 Notification Services

A straightforward approach to information dissemination is to use messaging middleware, typically through JMS, as transport protocol instead of HTTP. Messaging is often faster than HTTP [13], and furthermore more suited for those scenarios where asynchronous event notifications must be propagated to several destinations [27]. Unfortunately, this approach severely limits the ability to deploy across organizational boundaries and still does not meet throughput stability requirements.

At the present time, there are some protocols that provide the publish/subscribe message exchange pattern to Web Services. WS–Events was created by Hewlett-Packard in 2003, but has since been made obsolete both by WS–Eventing and WS–Notification, which are similar [17, 22]. WS–Eventing has two versions [17] that were released in 2004, and remains, at the present time, as a W3C draft [9] submitted in 2006.

WS–Notification is a family of three specifications (WS–BaseNotification [7], WS–BrokeredNotification [8] and WS–Topics [10]) whose latest version is 1.3, released in 2006. The previous versions of the standard were interconnected with WS–ResourceFramework, as these two specifications depended on each other [17]. This connection was disrupted in the last version of WS–Notification.

Although WS–BaseNotification allows the reception of *raw* notifications, i.e. through the invocation of an operation defined in the WSDL of the consumer, it also implies that additional logic must be added to the receiving applications, so they perform the subscription step, as well as on the emitting applications, so each consumer receives the adequate type of notification, according to its preferences enclosed in the subscription.

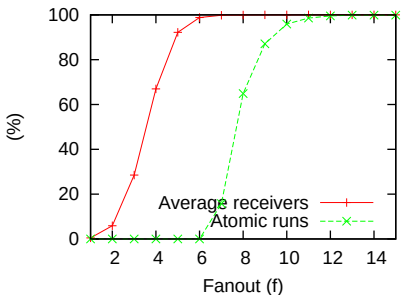

Figure 1: Reliability of gossip (1000 participants, 1000 dissemination runs, variable fanout).

Multiple implementations of these specifications exist. Apache ServiceMix [2] is an Enterprise Service Bus (ESB), built on the Java Business Integration (JBI) specification [6], which provides an implementation of version 1.3 of the WS–Notification standard. However, this implementation has some drawbacks, like the impossibility of clustering or persistent subscriptions, as well as the unrestricted message publishing, which can be performed by any node, even if it is a non-registered publisher.

WS-Messenger [3] implements both WS–Notification and WS–Eventing, and provides interoperability between the two specifications. However, the implemented versions are not the latest, standardized by OASIS, but, instead, the first specifications [5, 4] defined by consortiums of companies. Among other types of underlying messaging system, it also supports an implementation of JMS.

2.2 Gossip-based Dissemination

In a gossip or epidemic protocol, all the processes that make part of a system are potential disseminators of messages. Briefly, every process chooses randomly a subset of the remaining processes to which the message is then forwarded. Each of these processes behaves exactly in the same way when it receives a message. There is no *reactive* mechanism to deal with failures. This also mimics how epidemics spread in populations, hence the name epidemic protocols. Key parameters are:

Fanout (f) Number of targets that are locally selected by each process for gossiping.

Rounds (r) Maximum number of times a message is forwarded before being ignored.

The reliability of these algorithms is based on a *pro-active* mechanism where redundancy and randomization are used to avoid potential process and network link failures. It has also been shown that parameters f and r can be configured [16] such that any desired average number of receivers successfully get the message. Better yet, parameters can be set such that the message is atomically delivered to receivers with high probability.

Figure 1 illustrates this, by showing simulation results of disseminating 1000 messages to 1000 receivers, with $r = 5$ and a variable f. Notice that with $f > 6$ each destination gets each message with a very high probability. With $f > 11$, each message is atomically received by all destinations also with a very high probability. This happens even in face of process crashes and network faults.

The key to scalability is that the required fanout configuration is at worst logarithmically proportional to system size. Furthermore, it isn't required that nodes have perfect knowledge of the entire membership to select gossip targets. Instead, a small local system view built using the gossip protocol itself works as well [18].

There are however multiple variants of gossip protocols [20, 24], which provide different message exchange patterns and performance trade-offs:

Push Gossip A node that knows of new information, conveys it immediately to target nodes.

Lazy Push Gossip Optimizes the previous variant by deferring the transmission of the payload. A node that knows of some new data sends only the information topic. An interested node contacts another and, by sending the information topic, identifies the desired data. If the contacted node already has it, it just passes it through to the interested node. Otherwise, the originator of that data will complete the transmission. Similarly to the other lazy variant, this one is also useful when the data payload is very large, but also when it is very likely that the data is already known throughout the network.

Pull Gossip Instead of gossiping upon arrival of new information, a node periodically selects a number of peers and asks them for new information.

2-Phase Pull Gossip Very similar to pull gossip but where the target node sends only the recent information topic which must be asked for, explicitly by an interested node.

It has been shown that combining push and pull gossip dissemination is achieved in a lower number of steps [20]. The combination of both push gossip variants [24] achieves better performance in heterogeneous networks.

Gossip variants can have two alternative models [16], which differ in the behavior of the infected nodes. In the *infect--and-die* model, a node that is infected, i.e. receives a message, takes only one round to send the received message to other nodes, and then never sends it again, becoming dead in the analogy with nature. In the *infect-forever* model, also known as *balls-and-bins* [21], a node does not die, which means it can send a received message for more than one round, possibly until some stoppage criterion is satisfied.

Futher variations of the basic gossip procedure address the publish/subscribe model, in which messages are disseminated only to a subset of interested nodes, and multiple security concerns.

3. GOSSIP SERVICE

3.1 Assumptions

Figure 2 describes the architecture that is assumed in the information dissemination system where gossiping is to be applied. The list of subscribers is embedded in some application App0 which initiates message dissemination by invoking some operation *op()* in all of them. Interested applications, e.g., App1, request to be added to the list of subscribers by using some operation *subscribe()*.

The main assumption is thus that we consider operations without any return message, that is the same to say those

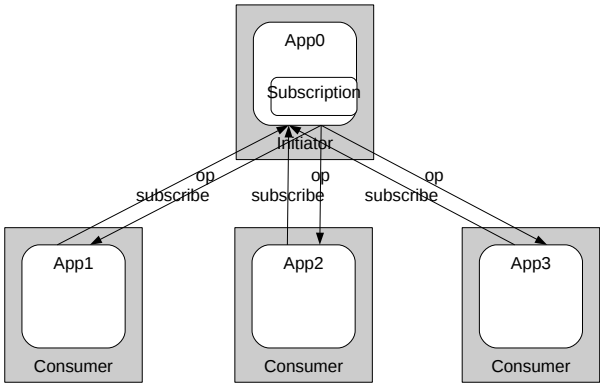

Figure 2: Typical dissemination scenario.

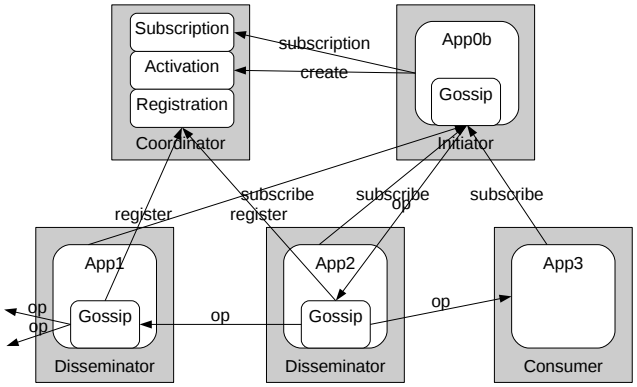

Figure 3: Dissemination using the gossip service.

that use an *In Only* message exchange pattern. This way the initiator of the gossip interaction can send the message and not bother about it again, in an asynchronous *fire and forget* manner.

Notice that it is irrelevant whether App0 generates information by itself or simply acts as a broker, relaying messages received by some other service. Note also that the assumed system fits both *ad hoc* dissemination applications as well as those built on top of standard notification services described in Section 2.1.

3.2 Architecture

The proposed WS–PushGossip service is built on the standard WS–Coordination framework in order to provide gossip-based communication seamlessly to any regular service that wishes to disseminate any invocation or result. Figure 3 presents an overview of its architecture. There are now four different roles:

Initiator Initiates the dissemination of each data item. This role requires that the application code (App0b) is changed to use the gossip service and that a compliant middleware stack is used.

Disseminator A node that receives a message, sends it to the peers in the list obtained from the Membership service. Although the application code is oblivious to the gossip service, a compliant middleware stack is required.

Consumer A node that receives a message, consumes it. This node is completely unchanged and unaffected by the introduction of gossip.

Coordinator Besides the Activation and Registration services from WS–Coordination, these nodes manage the subscription list.

The main impact of adopting WS–PushGossip is changing the initiator application to delegate subscription management and to issue a single notification, after having activated a gossip interaction with the Activation service. The middleware stack intercepts the outgoing message and reroutes it to selected destinations, i.e. App2 in Figure 3.

Upon arrival to App2, the message is again intercepted by the Gossip layer in the middleware stack. If this is an unknown gossip interaction, it registers itself with the Registration service, thus obtaining gossip targets to which it will forward the message. In the case of Figure 3, App1 and App3. The Coordinator can later refresh peer lists that were sent during registrations by sending messages to peers.

Assuming a single instance of the Coordinator for simplicity, it knows the entire list of subscribers, as well as those that are participating in gossiping. It is thus capable of providing adequate parameter configurations and peers for each gossip round.

Notice that a distributed Coordinator is supported by WS–Coordination and thus also by WS–PushGossip, as the list of subscribers can be maintained in a distributed fashion as proposed by WS–Membership [29]. It however an open issue if a distributed coordinator is really required, as long as managing membership is not a performance or dependability bottleneck. This is precisely where gossip-based protocols within a service-oriented computing setting might differ from previous applications of gossip, where such requirement was indeed true.

3.3 Operational Details

Figure 4 shows the header of a SOAP message, where the element *CoordinationContext* contains all the necessary information for a gossip interaction to be performed. Among that information we can find:

- the type of coordination, that in this case points to the WS–PushGossip namespace;
- the address of the Registration service of the Coordinator that created this context;
- the parameters used in the gossip mechanism, described in Section 2.2.

A message with such an header will be sent firstly by the Initiator of the gossip interaction, and this only occurs after the Initiator has contacted the Coordinator to create a gossip context. The operations that lead to this creation are depicted in Figure 5 and described as follows. The Initiator sends a message to the Coordinator invoking the *CreateCoordinationContext* operation, with the desired values for the gossip parameters as arguments. Then, the Coordinator creates the coordination context with the indicated parameters, and sets the type of coordination to gossip, and the address of the Registration service to point to its own. After this, the Coordinator retrieves the list of peers of the Initiator from the Membership service, giving knowledge of

```
<?xml version="1.0" encoding="utf-8"?>
<S11:Envelope
 xmlns:S11="http://www.w3.org/2003/05/soap-envelope">
 <S11:Header>
  . . .
  <wscoor:CoordinationContext
   xmlns:wsa="http://www.w3.org/2005/08/addressing"
   xmlns:wscoor=
   "http://docs.oasis-open.org/ws-tx/wscoor/2006/06"
   xmlns:wspg=
   "http://gsd.di.uminho.pt/ws/gossip/wspg/2008/06"
   S11:mustUnderstand="true">
   <wscoor:Identifier>
    Gossip1
   </wscoor:Identifier>
   <wscoor:Expires>3000</wscoor:Expires>
   <wscoor:CoordinationType>
    http://gsd.di.uminho.pt/ws/gossip/wspg/2008/06
   </wscoor:CoordinationType>
   <wscoor:RegistrationService>
    <wsa:Address>
     http://gsd.di.uminho.pt/ws/gossip/my/reg
    </wsa:Address>
    <wsa:ReferenceParameters>
     <wspg:Fanout>5</wspg:Fanout>
     <wspg:Rounds>3</wspg:Rounds>
    </wsa:ReferenceParameters>
   </wscoor:RegistrationService>
  </wscoor:CoordinationContext>
  . . .
 </S11:Header>
 <S11:Body>
  . . .
 </S11:Body>
</S11:Envelope>
```

Figure 4: SOAP message with the gossip coordination context.

the desired fanout which determines the size of the returned list. This list is then conveyed, alongside with the newly created context, in the same return message. After receiving this message, the Initiator can then send the message to the peers on the received list.

On Figure 6, the Initiator who created a gossip context, as depicted in Figure 5 (*s1*), sends the message to the services *s2* and *s3*.

When a Disseminator, like service *s2*, receives a message containing a gossip context, it knows it must register with the indicated Coordinator to intervene in the message propagation. The registration process is also depicted on Figure 6. It starts by the invocation of the *Register* operation on the Coordinator by the Disseminator. The received context is also transmitted so that the Coordinator can know to which gossip interaction the Disseminator wants to become part of. The Coordinator then retrieves and sends to the Disseminator its list of peers. Then, the Disseminator is able to propagate the message to its peers, and this set of operations is repeated by each peer that performs the Disseminator role.

On the other hand, when a Consumer, like service *s3*, receives a similar message, it just consumes its contents, that is

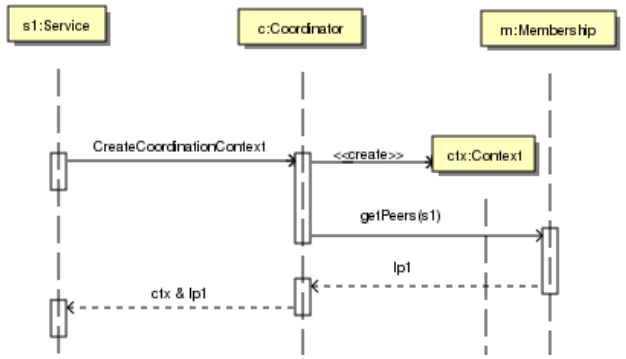

Figure 5: Sequence diagram of the creation of a gossip context

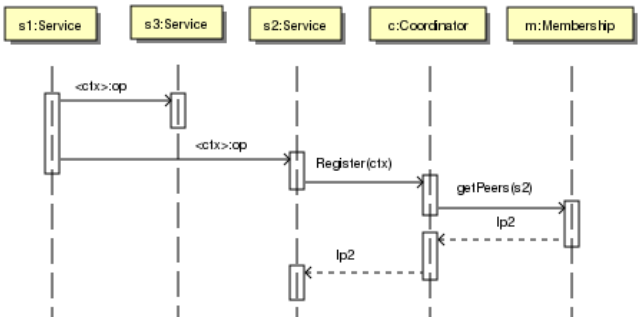

Figure 6: Sequence diagram of the registration of a node to a gossip interaction

the same to say, that it processes internally the information contained on the body of the message, without resending it or intervening in any way in the gossip interaction.

To terminate a gossip interaction, the expiration time included in the context should suffice. However, an alternative mechanism may be used by any Disseminator. This mechanism relies on the invocation of the *Finish* operation on the Coordinator, so it will discard the gossip interaction that corresponds to the context conveyed as a parameter.

4. IMPLEMENTATION

Apache Axis2 is a Web Services platform which has matured with the acquired experience in Apache SOAP and Apache Axis1. The Axis2 engine is a pure SOAP processor. It receives a message through some transport, calls the predefined set of handlers to process the message, which is then delivered to a Message Receiver. The Message Receiver usually hands over the message to the service implementation class for processing.

We implemented our protocol as functional Web Service extensions allowed by the Axis2 handler framework. An handler normally processes information inside the SOAP headers, but it isn't restricted to process other parts of a message. However, to fully understand the concept of an handler or interceptor, we must define what a flow and a phase represent in the Axis2 architecture. The phase concept was introduced in Axis2 to easily extend core functionality. A phase is a collection of one or more handlers, which are related and connected according to the phase rules. The name of each phase represents what happens to a message during it. Similarly to the previous concept, a flow or pipe is a collection of phases, and it can be classified as inflow, when receiving a message, or outflow, when sending a message.

Axis2 introduced the concept of module to provide additional functionality which, compared to Axis1, has the advantage of not requiring any changes to global configuration files. A module contains handlers, third-party libraries, needed resources and a configuration file, which specifies handlers and their phase rules. A module is said to be available once it is put into the repository, but it only becomes active or engaged by adding an entry to axis2.xml or by using the Axis2 administration web interface. Only then, the handlers of the module are added to the flows.

The implementation of this framework as an Axis2 module, allows existing Axis2 SOAP engines to be upgraded in order to use it by simply loading it before any gossip interaction.

Gossip interactions must not be restricted to Axis2 engines that have loaded the WS–PushGossip module. In that sense, other SOAP engines, other than Axis2, and even Axis2 SOAP engines which have not loaded the module, can intervene in the interaction with a minor role, as they simply become recipients for the messages.

5. CONCLUSION

It is a well known problem in distributed systems that stable high throughput when disseminating information in large scale heterogeneous systems is a hard problem. Current state-of-the-art points towards gossip-based protocols as the best option regarding scalability and resilience.

In this paper we point out that interesting service-oriented architectures, namely, those based on recently proposed notification services, will face the same difficulties regarding resilience and scalability.

Towards leveraging gossip-based protocols as a high-level structuring paradigm that inherently avoids such problems, we make the contribution that message dissemination done in a distributed fashion by gossiping can be regarded as a coordination problem. As a proof-of-concept we then propose WS–PushGossip, based on the standard WS–Coordination framework, that illustrates the approach.

Current work is focused on achieving a gossip-based service framework that encompasses further a wider range of gossip styles and provides complete functionality, namely, by fully integrating in the Web Service ecosystem. This will require closely evaluating each design and implementation decision.

6. REFERENCES

[1] Apache Axis2 Homepage. `http://ws.apache.org/axis2/`.

[2] Apache ServiceMix. `http://servicemix.apache.org/home.html`.

[3] WS-Messenger. `http://www.extreme.indiana.edu/xgws/messenger/`.

[4] WS-Eventing Specification. `http://www.ibm.com/developerworks/webservices/library/specification/ws-%eventing/`, 01 August 2004.

[5] WS-Notification Specification. `http://www.ibm.com/developerworks/webservices/`

library/specification/ws-%notification/, 01 March 2004.

[6] JSR 208: Java Business Integration (JBI). http://www.jcp.org/en/jsr/detail?id=208, 25 August 2005.

[7] WS-BaseNotification 1.3 OASIS Standard. http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf%, 1 October 2006.

[8] WS-BrokeredNotification 1.3 OASIS Standard. http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os%.pdf, 1 October 2006.

[9] WS-Eventing W3C Member Submission. http://www.w3.org/Submission/WS-Eventing/, 15 March 2006.

[10] WS-Topics 1.3 OASIS Standard. http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf, 1 October 2006.

[11] K. Birman. A review of experiences with reliable multicast. *Software Practice and Experience*, 29(9), July 1999.

[12] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[13] R. Eggen and S. Sunku. Efficiency of Soap Versus JMS. In *International Conference on Internet Computing*, pages 99–105, 2003.

[14] A. Erradi, P. Maheshwari, and V. Tosic. WS-Policy based Monitoring of Composite Web Services. *Web Services, 2007. ECOWS '07. Fifth European Conference on*, pages 99–108, Nov. 2007.

[15] A. Erradi, V. Tosic, and P. Maheshwari. MASC - .NET-Based Middleware for Adaptive Composite Web Services. *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 727–734, July 2007.

[16] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, May 2004.

[17] Y. Huang and D. Gannon. A comparative study of Web services-based event notification specifications. *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 8 pp.–, Aug. 2006.

[18] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[19] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast Computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Nov. 2003.

[20] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 565–574, 2000.

[21] B. Koldehofe. Simple gossiping with balls and bins. In *Proceedings of the 6th International Conference on Principles of Distributed Systems (OPODIS'02)*, pages 109–118, 2002.

[22] G. Monsieur, M. Snoeck, and W. Lemahieu. Coordinated Web Services Orchestration. *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 775–783, July 2007.

[23] G. S. Niblett, P. Events and service-oriented architecture: The OASIS Web Services Notification specifications. *IBM Systems Journal*, 44(4):869–886, 25 October 2005.

[24] J. Pereira, R. Oliveira, and L. Rodrigues. Efficient Epidemic Multicast in Heterogeneous Networks. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4278/2006, pages 1520–1529. Springer Berlin / Heidelberg, October 2006.

[25] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange trading system. *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 309–313, Jun 1997.

[26] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[27] E. Roch. Web Services HTTP vs. JMS. http://it.toolbox.com/blogs/the-soa-blog/web-services-http-vs-jms-19110%, 17 September 2007.

[28] W. Vogels. All Things Distributed. http://www.allthingsdistributed.com/.

[29] W. Vogels and C. Re. WS-Membership - Failure Management in a Web-Services World. 2003.