Byzantine Fault-Tolerant Publish/Subscribe: A Cloud Computing Infrastructure

(Position Paper)

Tiancheng Chang and Hein Meling
Department of Electrical Engineering and Computer Science
University of Stavanger, Norway
{*tiancheng.chang, hein.meling*}*@uis.no*

***Abstract*—The emerging publish/subscribe communication paradigm for building large-scale distributed event notification systems, has been shown to exhibit excellent performance and scalability characteristics. Moreover, some work also focus on providing reliability and availability guarantees in the face of node crash and link failures. Such publish/subscribe systems are commonly used in cloud computing infrastructures. However, addressing the dependability concern due to malicious attacks or unintentional software errors, which can potentially corrupt the system, has largely been left untouched by researchers.**

In this paper, we first identify some of the potential problem areas related to Byzantine behavior in the publish/subscribe paradigm. Secondly, we propose several directions of research for designing a Byzantine fault-tolerant publish/subscribe system suitable for use as a cloud computing infrastructure.

I. Introduction

The ever increasing scale of geographically dispersed systems is severely challenging communication infrastructure designers. In particular, the traditional tightly-coupled and synchronous client/server model is being challenged by the evolving publish/subscribe (pub/sub) interaction paradigm. An appealing trait of pub/sub is its ability to handle wide-area data propagation across an enormous number of data sources and sinks. Messages are typically forwarded from *publishers* to *subscribers* through *brokers*. Brokers are responsible for matching publications to subscribers and forwarding messages accordingly. In its simplest form, one broker acts as the centralized service provider which joins all publishers and subscribers. A modish approach is to use a set of dedicated brokers to form an overlay network.

Pub/sub is an important cloud computing infrastructure widely adopted in industry, e.g. Windows Azure service bus [1], Google GooPS [2], Oracle Java Messaging Service [3], and IBM WebSphere [4]. A significant effort has been devoted to developing high-performance and highly scalable pub/sub systems [5], [6], as well as crash failure resilience [7]. However, to our knowledge, very little effort has gone into developing dependable pub/sub systems that can tolerate Byzantine faults [8]. A Byzantine faulty node may deviate from the protocol specification and can cause the pub/sub system to enter a corrupted or arbitrary system state, which can have serious consequences for users of the system.

In this paper, we sketch a Byzantine fault-tolerant cloud infrastructure for pub/sub systems. The infrastructure aims to ameliorate the malicious behavior by brokers, publishers, and subscribers, in addition to normal crash failures. For example, a Byzantine faulty broker could falsify messages *en route* to subscribers, which would cause damage to system integrity, e.g. in a stock system, it is unacceptable for subscribers to receive falsified or even delayed stock quotes. Besides, malicious publishers and subscribers could invoke a denial-of-service attack on the system by sending vast amounts of publications or subscription requests. Our proposed system aims to offer a cloud computing infrastructure which ensures system safety and achieves liveness despite Byzantine failures, by masking and eventually excluding Byzantine faulty nodes.

II. Scope and Assumptions

In our envisioned cloud infrastructure, we assume topic-based pub/sub [9] as our starting point. In a topic-based system, messages are published to topics, and subscribers receive all messages published to the topics to which they subscribe; all subscribers to a given topic receive the same messages. We do not assume advertisement-based routing at this point in time. Advertisements are generally used to optimize content-based pub/sub, and are produced by publishers to announce their intention to publish certain types of publications. Our decision to focus on topic-based pub/sub is for simplicity, however, content-based pub/sub will be considered in future work.

We are interested in both centralized and distributed pub/sub architectures. In a centralized architecture, a single service provider is used. Thus, replication is necessary to build a fault-tolerant centralized pub/sub system. Moreover, a distributed pub/sub architecture is comprised of a set of cooperating brokers that form an overlay network. In pub/sub, we distinguish between three different agent roles: *subscriber*, *publisher*, and *broker*. In addition to a purely Byzantine failure model, we are interested in different combinations of failure models, in which the different agent roles may assume different failure models. For example, we may consider a system in which brokers only tolerate benign crash failures among themselves, while still being resilient to Byzantine behavior of the external agents, i.e. publishers and subscribers [10]. Thus, in this paper, we identify misbehaviors based on agent roles.

We assume asynchronous communication. Messages can be delayed indefinitely, thus a failure detector cannot distinguish between a node crash and link failure.

III. Problem Analysis: Failure Scenarios

Inspired by analysis of attacks due to misbehaving routers [11], denial-of-service attacks by publishers and subscribers [12], and other malicious behaviors and threats in pub/sub [13], [14], we enumerate the following potential misbehavior scenarios:

(*a*) Broker: A Byzantine broker's misbehaviors include:

- Publication delay: A Byzantine faulty broker can inject arbitrary delays. This can reduce performance, cause omission failure, or even system outage in the worst case;
- Message reordering: A faulty broker can order publications and subscriptions arbitrarily, violating ordering principles implemented in the system. This can cause performance issues or even denial-of-service for subscribers;
- Subscription corruption: A faulty broker can modify, drop, or inject subscriptions. This will contaminate routing tables, which may cause publication loss or erroneous forwarding of publications to unauthorized subscribers;
- Publication corruption: A faulty broker can corrupt a publication by tampering with its content, and it can also inject arbitrary publications. This behavior may lower system integrity for subscribers.

(*b*) Publisher: A Byzantine publisher's behaviors include:

- Spam injection: A faulty publisher can inject spam publications which draw no interest from subscribers. This can be done by colluding with malicious brokers. It will cause internal performance problem for benign brokers or even denial-of-service;
- Publication flooding: A faulty publisher can produce publications to match as many publications as possible. This will affect service quality for subscribers;
- Colluding attack: A faulty publisher can collude with malicious subscribers to generate overwhelming amounts of publications. This causes denial-of-service to brokers.

(*c*) Subscriber: A Byzantine subscriber's behaviors include:

- Greedy subscription: A faulty subscriber can greedily subscribe to all types of publications. This kind of behavior is generally not forbidden in previous research works. But it can lead to overwhelming traffic;
- Frivolous subscription: A faulty subscriber can repeatedly subscribe and unsubscribe to produce a large number of routing table update traffic; this will affect the service quality of brokers;
- Deny acknowledgement: A faulty subscriber can deny receiving publications. This will prevent the responsible brokers from garbage collection and cause retransmission in pub/sub systems requiring acknowledgement;
- Colluding attack: A faulty subscriber can collude with publishers to receive large amounts of publications.

In addition to the above malicious faults, nodes may also crash.

To build a fault-tolerant pub/sub system capable of tackling the above mentioned problems is not an easy task. For example, in (*a*) it is difficult, maybe even impossible, to differentiate malicious publication delays from delay due to asynchrony of links; for (*b*) it is difficult to say whether a publisher is benignly generating lots of publication or maliciously flooding the system with spam; in (*c*) it is not easy to separate a subscriber's denial to receive (acknowledge) certain publications and actual publication loss caused by link or node failure.

Our goal is build a system that will cover as many of these Byzantine faulty behaviors as possible.

IV. Byzantine Fault-Tolerant Pub/Sub

Byzantine fault tolerance in a generic point-to-point architecture has reached a certain level of maturity, and usually rely on the Replicated State Machine (RSM) approach. The RSM is a general approach for constructing fault-tolerant services, and a key protocol underlying RSM is consensus [15]. Several Byzantine fault-tolerant (BFT) consensus protocols [16], [17] have been used to construct client/server systems that need to resist the effects of compromise.

We propose to use replication techniques, primarily for brokers, in our infrastructure to mask the effects of Byzantine failures. Our decision is based on the fact that, it is impossible for a single entity to determine that its neighbor node is behaving abnormally. As in the traditional consensus problem, such detection demands synchronization between a quorum of correctly behaving nodes. BFT techniques have also been used in another domain, namely to facilitate a Byzantine tolerant tuple space [18]. A tuple space is a essentially a distributed shared memory, which enables clients to conduct decoupled coordination. A tuple space is essentially based on a client-server architecture, and aims to facilitate data sharing. This is different from the objective of event-based pub/sub, which is to provide a message passing and filtering architecture. Unlike globally-accessible data, events produced in a pub/sub system is only delivered to entities that have subscribed to it. Moreover, in some cases, the event may only be observable for the duration of message passing to its presently available subscribers [19]. In other cases, a message queue may hold events for delivery to new subscribers.

The nature of pub/sub makes it very different from the traditional client-server approach usually considered in BFT systems. We aim to study replication in the following forms of pub/sub infrastructure: (*i*) centralized replicated brokers; and (*ii*) overlay of replicated brokers. The two forms of replication correspond to the two types of pub/sub architectures.

A. Centralized Replicated Brokers

We propose an RSM-based centralized pub/sub architecture, in which a set of replicated brokers are used to offer a centralized service. The set of replicated brokers appears to the publishers and subscribers, to be one virtual broker, offering transparent service. With this strategy, we can avoid system outage caused by the single point of failure as long as a quorum of brokers remains live. We can also avoid that the system becomes compromised or enters an arbitrary state, by masking the behavior of Byzantine faulty brokers[1] inside the set of replicated brokers, including corrupting publications, modifying routing

[1]Masking f Byzantine brokers requires $3f + 1$ replicas.

table entries, arbitrary delay, message dropping and reordering, as long as there is a quorum of non-faulty brokers in the set.

There remains a number of challenges in applying BFT to pub/sub: (*A*) generic BFT protocols are often criticized for performance and latency issues, and high replication degree. It remains an open question how these protocols can be optimized for pub/sub; (*B*) previous research on BFT protocols relies on time and synchronization coupling to achieve liveness. However, an important trait of pub/sub is its decoupled nature, which may demand new protocols; (*C*) since a cloud infrastructure for pub/sub is expected to be large-scale and wide-area, it becomes prudent to question the scalability of the centralized replicated broker architecture. One approach is use an overlay of replicated brokers, as we discuss next.

B. Overlay of Replicated Brokers

Often brokers may be interconnected over some overlay network [20] to scale the number of publishers, subscribers and the number of events that can be handled. Moreover, an overlay network may also serve as a basis for providing fault-tolerant communication. Several network topologies are possible. A linear topology offers the simplest method to add new nodes by serially attaching each node to the next. But it is generally expensive in terms of computing and lacks in fault tolerance. The mesh topology (fully-connected), is usually complex and expensive unless deployed on a small set of nodes. A tree-based topology is most frequently used, because it can efficiently connect a large number of nodes.

As a method of scaling up the centralized infrastructure, we can (*1*) use a small set of replicated brokers to represent a single broker in the overlay. In the traditional approach, failure of an intermediate broker causes propagation interruption or compromise unless the faulty broker is recovered or bypassed. Replication of brokers prevents that, as long as each set of replicated brokers retains liveness and safety; and (*2*) use replicated propagation paths [13] as a second barrier. Usually, there is only one path between two neighboring brokers. And thus failures require topology reconfiguration to resume. By replicating propagation paths, we avoid reconfiguring topology, as long as at least one of the replicated paths is active.

Yet, there remains numerous challenges. It is difficult to plainly implement replication on any type of the topologies considering that replication increases interconnection complexity thus the replicated topology will be costly. Besides, replicated propagation paths occupy more network resources. It remains a question of organizing a suitable topology to neutralize the interconnection complexity raised by replication.

C. Suspicion Mechanism

Apart from masking Byzantine faulty brokers through use of replication, mechanisms are also needed to mask malicious behaviors by publishers and subscribers. However, it is difficult to distinguish malicious from benign behaviors.

To address this problem, we let the *edge brokers* implement suspicion and incentive mechanism revised for pub/sub. Namely, a publisher/subscriber's inappropriate behaviors, e.g. delay, message loss or overwhelming, will be suspected and punished to some extent. The punishment ensures that misbehaviors will eventually be no cheaper than acting in accordance with the original protocol. When the inappropriate behaviors by a publisher or subscriber exceed certain thresholds, e.g. reacting too slowly or too frequently than our predefined parameter, we mask it as malicious and exclude. Such suspicion and punishment mechanism was also used in [21].

V. Summary

Byzantine fault tolerance in cloud computing infrastructures such as pub/sub has yet to be sufficiently addressed. We aim to develop a Byzantine fault-tolerant protocol for pub/sub, and subsequently a prototype system.

Acknowledgement

Thanks to Leander Jehl and Roman Vitenberg for fruitful discussions and feedback.

References

[1] T. Redkar, *Windows Azure Platform*. Apress, 2010.

[2] J. Reumann, "Pub/sub at google," *CANOE Summer School*, 2009.

[3] R. Monson-Haefel and D. Chappell, *Java Message Service*. O'Reilly & Associates, Inc., 2000.

[4] F. Budinsky, G. DeCandio, R. Earle, T. Francis, J. Jones, J. Li, M. Nally, C. Nelin, V. Popescu, S. Rich, A. Ryman, and T. Wilson, "Websphere studio overview," *IBM Syst. J.*, vol. 43, no. 2, pp. 384–419, Apr. 2004.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, no. 3, pp. 332–383, Aug. 2001.

[6] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *ICDCS*, 1999, pp. 262–272.

[7] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *SRDS*, 2009, pp. 41–50.

[8] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, July 1982.

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, June 2003.

[10] H. Meling, K. Marzullo, and A. Mei, "When you don't trust clients: Byzantine proposer fast paxos," in *ICDCS*, 2012.

[11] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage, "Detecting and isolating malicious routers," *IEEE Trans. Dependable Secure Comput.*, vol. 3, pp. 230–244, 2006.

[12] A. Wun, A. Cheung, and H.-A. Jacobsen, "A taxonomy for denial of service attacks in content-based publish/subscribe systems," in *DEBS*, 2007, pp. 116–127.

[13] Private communication.

[14] T. R. Mayer, L. Brunie, D. Coquil, and H. Kosch, "Evaluating the Robustness of Publish/Subscribe Systems," in *3PGCIC*, 2011.

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[17] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *SOSP*, 2007, pp. 45–58.

[18] A. N. Bessani, E. P. Alchieri, M. Correia, and J. da Silva Fraga, "Depspace: A byzantine fault-tolerant coordination service," in *ACM SIGOPS/EuroSys*, Apr. 2008.

[19] N. Busi and G. Zavattaro, "Publish/subscribe vs. shared dataspace coordination infrastructures: Is it just a matter of taste?" in *WETICE*, 2001.

[20] M. van Steen, *Graph Theory and Complex Networks: An Introduction*. Maarten van Steen, 2010.

[21] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "Bar fault tolerance for cooperative services," in *SOSP*, 2005, pp. 45–58.