P2S: A Fault-Tolerant Publish/Subscribe Infrastructure

Tiancheng Chang
University of Stavanger, Norway
tiancheng.chang@uis.no

Sisi Duan
University of California, Davis
sduan@ucdavis.edu

Hein Meling
University of Stavanger, Norway
hein.meling@uis.no

Sean Peisert*
University of California, Davis
peisert@cs.ucdavis.edu

Haibin Zhang
University of California, Davis
hbzhang@ucdavis.edu

ABSTRACT

The popular publish/subscribe communication paradigm, for building large-scale distributed event notification systems, has attracted attention from both academia and industry due to its performance and scalability characteristics. While ordinary "web surfers" typically are not aware of minor packet loss, industrial applications often have tight timing constraints and require rigorous fault tolerance. Some past research has addressed the need to tolerate node crashes and link failures, often relying on distributing the brokers on an overlay network. However, these solutions impose significant complexity both in terms of implementation and deployment.

In this paper, we present a crash tolerant Paxos-based pub/sub (P2S) middleware. P2S contributes a practical solution by replicating the broker in a replicated architecture based on Goxos, a Paxos-based fault tolerance library. Goxos can switch between various Paxos variants according to different fault tolerance requirements. P2S directly adapts existing fault tolerance techniques to pub/sub, with the aim of reducing the burden of proving the correctness of the implementation. Furthermore, P2S is a development framework that provides sophisticated generic programming interfaces for building various types of pub/sub applications. The flexibility and versatility of the P2S framework ensures that pub/sub systems with widely varying dependability needs can be developed quickly.

We evaluate the performance of our implementation using event logs obtained from a real deployment at an IPTV cable provider. Our evaluation results show that P2S reduces throughput by as little as 1.25% and adds only 0.58 ms latency overhead, compared to its non-replicated counterpart. The performance characteristics of P2S prove the feasibility and utility of our framework.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*distributed systems*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.2.11 [**Software Architectures**]: Patterns

*S. Peisert is also with Lawrence Berkeley National Laboratory.

DEBS '14, May 26-29, 2014, Mumbai, India.

General Terms

Experimentation, Measurement, Performance

Keywords

Fault tolerance, Paxos, Publish/Subscribe, IPTV Application

1. INTRODUCTION

The publish/subscribe communication pattern for constructing event notification services has strong performance and flexibility characteristics. While typical "pub/sub" services such as consumer RSS news feeds may tolerate some level of message loss, enterprise applications often demand stronger dependability guarantees. As a result, pub/sub has become an important cloud computing infrastructure and is widely used in industry, e.g., in Google GooPS [1], Windows Azure Service Bus [2], Oracle Java Messaging Service [3], and IBM WebSphere [4].

Significant effort has been devoted to developing reliable pub/-sub systems [5–13]. Most of them cope with broker crashes and/or link failures, ensuring that messages are eventually delivered. While the weak fault tolerance is sufficient in some systems, other application domains demand stringent delivery order of their messages. Only a handful of prior published research papers have discussed how to achieve total ordering in reliable pub/sub systems [8–10]. In order to guarantee total ordering in the presence of failures, virtually all past published work relies on an overlay network topology. For each new type of topology, a different algorithm must be introduced, adding significant complexity both in terms of algorithm correctness proofs, implementation, and deployment. Therefore, industrial deployments tend to rely on the more established centralized architecture instead of decentralized overlay topologies.

Traditional fault tolerance techniques based on Paxos [14] can provide total ordering and guarantee safety even in the presence of any number of failures. However, liveness cannot be ensured in periods of asynchrony. Building a reliable pub/sub system based on an existing, proven approach, reduces the effort required to prove the correctness of algorithms since the protocol can be proven correct by refinement from the original algorithm. However, adapting traditional fault tolerance techniques to pub/sub systems is challenging. Intuitively, every broker can be replicated, which can be extremely impractical. Total ordering on every message can be overkill since different messages may require different ordering semantics. For instance, per-publisher total ordering is sufficient for publications from a single publisher to multiple subscribers. On the other hand, the topology of brokers in pub/sub systems varies from a single centralized broker to very large-scale overlays. Replication of brokers may impose adjustment of pub/sub overlays, especially

when the brokers are replicated on demand. Therefore, management of replication should impose minimum overhead.

In this paper, we propose a framework for building reliable pub-/sub systems that directly adapts existing fault tolerance techniques to pub/sub. At the core of our pub/sub infrastructure is our crash fault tolerance library and a pub/sub interface. Our library guarantees fault tolerance through replication, and ensures strong consistency using Paxos to order publications. Our fault tolerance library can switch between different consistency protocols depending on application specific fault tolerance requirements. On the other hand, the pub/sub interface communicates between application level roles (publishers, subscribers, and the brokers) and the replication library. The interface takes publications that must be totally ordered, and pass them on to the replication library as requests and totally orders them. The messages are then delivered to the corresponding subscribers in order.

We have designed P2S, a topic-based crash tolerant pub/sub system based on a replication library Goxos [15, 16], a Paxos-based Replicated State Machine (RSM) [17] framework written in the Go programming language [18]. P2S is motivated by the simplest pub-/sub architecture that is employed in several industry settings: publishers and subscribers with only a centralized broker. Since the centralized broker becomes a single point of failure, we replicate the broker to achieve resilience. To ensure total ordering, a Paxos-based library is run among the replicated brokers. Although we adopt the architecture of P2S directly from existing fault tolerance protocols, we are not aware of any other published work discussing the implementation of such solutions and therefore the performance characteristics have previously not been explored and published. We further evaluate the performance of P2S using recorded event logs obtained from a real deployment of event loggers at about 180,000 homes connected to an IPTV cable provider. Our evaluation results show that P2S causes as low as 1.25% reduction in throughput and only 0.58 ms end-to-end latency overhead compared to its non-replicated counterpart.

Our paper makes the following key contributions:

1. We implemented P2S, the simplest architecture based on the framework, a topic-based crash tolerant pub/sub system with centralized replicated brokers.
2. We demonstrate the utility of P2S through experiments using recorded data logs obtained from an IPTV application deployed at a national telco operator. The evaluation results show that P2S achieves total ordering in the presence of failure with low overhead compared to its non-replicated overhead.
3. We present a framework for building reliable pub/sub systems that directly adapts existing proven fault tolerance approaches, with a relatively simple correctness proof and implementation. The framework is flexible and versatile enough to be used in future development.

The rest of the paper is organized as follows: first, we introduce some background for our work in §2. In §3, we depict design and development details of our framework. Then we show experimental results in §4. We present related work in §5 and conclude by reviewing our contributions in §6.

2. BACKGROUND

In this section we present background for our fault-tolerant pub/sub system, P2S. We begin by introducing Paxos, a well-known crash fault-tolerant consensus protocol on which we base P2S. We then briefly summarize the pub/sub architecture on which we base P2S.

2.1 Fault Tolerance

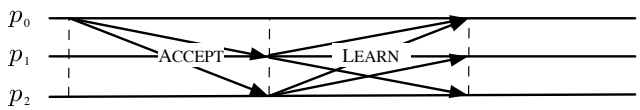
The Paxos protocol [14, 19] is a fault-tolerant consensus protocol, in which a set of participants (our replicas) try to reach agreement on a value. For our purpose, we can use multiple instances of Paxos to agree on a sequence of values (or commands) sent to an RSM. This is also called Multi-Paxos. With Paxos, the participants can reach agreement when at least $f+1$ of the participants are able to communicate, where f is the number of replica failures that can be tolerated. One of the nice properties of Paxos is that it guarantees that consistency among the replicas will never be violated even if more than f replicas fail. It achieves this property at the expense of liveness. That is, if more than f replicas fail, or if fewer than $f+1$ replicas are able to communicate, then Paxos cannot make progress. Ensuring strong consistency among replicas is an important property, useful for a wide range of systems, including pub/sub systems. This is related to the fundamental tradeoff between strong and weak consistency.

We now explain how one instance of Paxos might operate in the pub/sub paradigm. First suppose that the participants must be made to agree on a single value or command to execute on our broker RSM. This command can be considered as a publication. Thus, the following is concerned with only a single command/publication. Paxos is often explained in terms of two phases, where the first phase is only invoked initially and to handle failures, while the second phase represents the normal case operation, and must be performed for every value to be agreed upon.

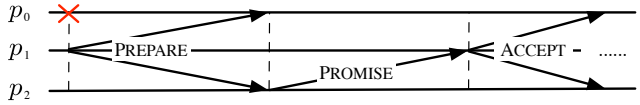
Paxos proceeds in rounds, where in each round there is a single replica designated as the *proposer*, also called the *leader*. Figure 1(a) depicts the normal case operation where the proposer is correct. During the normal case operation, the proposer chooses a value and sends an ACCEPT message to a set of replicas called *acceptors*. If an acceptor accepts the value, it sends an LEARN message to all the replicas. The value is chosen when a replica receives LEARN messages from a majority of replicas.

When the current proposer is suspected to be faulty, another replica may assume the role of proposer. To be effective as proposer, it needs to collect support from a majority of the replicas. It does so by broadcasting a PREPARE message to the other replicas. Upon receiving the PREPARE message, a replica stops accepting messages from the old proposer and replies to the new proposer with a PROMISE message, and includes the value chosen in its last round. When the leader collects a set of PROMISE messages from a majority of replicas, it either selects a value if at least one replica accepts it, or any value, if no replica includes any values in their PROMISE messages. Afterwards, replicas proceed as in normal case operation described above. Figure 1(b) shows the leader change phase of Paxos.

For a more comprehensible description of Paxos, please see [20].



(a) The normal-case operation



(b) Paxos leader/proposer change

Figure 1: The Paxos Protocol.

2.2 Pub/Sub

We build on the pub/sub architecture described by Eugster et al. [21], as illustrated in Figure 2. In a topic-based pub/sub system, *subscribers* express their interests in certain types of events, and are subsequently notified with *publications*, generated by *publishers*. *Brokers* are placed at the center of the infrastructure to mediate communication between publishers and subscribers. This event-based interaction provides full decoupling in *time*, *space*, and *synchronization* between publishers and subscribers. We assume topic-based pub/sub [21], where messages are published to topics, and subscribers receive all messages sent to the topics to which they subscribe.

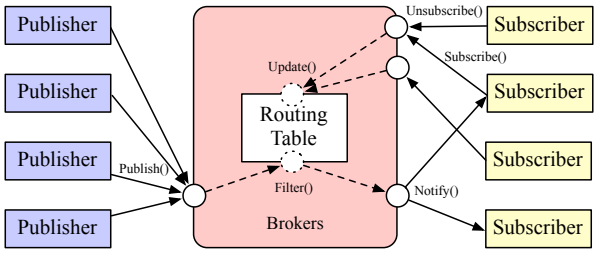


Figure 2: Publish/Subscribe architecture with three agent roles

In this paper, we address broker crash failures in an asynchronous model, where messages can be delayed, duplicated, dropped, or delivered out of order. P2S employs a simple pub/sub architecture: between publishers and subscribers is a set of $2f+1$ replicated brokers, among which up to f broker failures are tolerated. The replicated brokers can be in one or more administrative domains, perhaps geographically separated.

The protocol provides both safety and liveness as defined below. The safety property is also referred to as *total order*, which is defined in multiple ways in the pub/sub literature. For instance, *per-publisher total order* ensures that messages sent by a single publisher are totally ordered. Our system aims to achieve the strongest safety properties—*pairwise total order*—where replicated brokers behave like a centralized broker.

- (**Pairwise total order (Safety)**) Assume messages m and m' are delivered to both subscribers p and q, m is delivered before m' at p if and only if m is delivered before m' at q.
- (**Liveness**) If a message is delivered to a subscriber, all correct subscribers to the same topic eventually receive the same message.

3. P2S

Our P2S framework is built on our existing Paxos-based RSM library, Goxos [15, 16, 22]. For higher level pub/sub application builders, P2S provides a generic programming interface.

This section introduces details of the original Goxos implementation, along with some changes necessary in order to adapt Goxos to the pub/sub model. We also present the P2S system architecture, programming APIs, some application-specific implementation details, and the core broker algorithm that runs inside each P2S broker. Essentially, when messages are sent by clients (either publishers or subscribers) to brokers, they are handled by the Goxos library. Goxos treats client messages as Paxos requests, orders them accordingly, and delivers them to the broker application layer. The broker then forwards the messages to the subscribers according to the message type (topic).

3.1 Goxos Architecture and Implementation

Goxos provides the fault-tolerant library for P2S. That is, P2S implements Goxos interfaces to replicate its broker. As long as no more than f brokers fail, all failure handling is managed internally in the underlying Goxos framework in the manner that Paxos originally describes and will not be observed by publishers or subscribers. Thus, Goxos provides a much greater degree of tolerance to crashes compared to a traditional broker-based pub/sub system.

Using Goxos for replicating the broker, one of the replicas will serve as the leader to handle client requests. A client, either a publisher or a subscriber, first reads a configuration file that contains information about the location of the replicas. Using this information, the client connects to the first replica, which is usually the leader. If it is not the leader, the client attempts to connect to the next replica in the list. The leader receives the client's connection attempt, then establishes the connection, and stores the client connection for further interactions. The client then is able to send requests to either issue a publication, a subscription, or unsubscription to the leader. Upon receiving a valid client request, the leader treats the request as a Paxos proposal and disseminates it to all Goxos replicas to achieve consensus. Thus, each replica decides on the ordering of potentially several competing requests and then executes them in order. Then finally, the execution result is replied back to the client.

This original implementation does not match the pub/sub model because it acts strictly in the classic request-reply style. That is, it lacks the logic to handle message forwarding, which is necessary when a request, e.g. a publication, should be forwarded to subscribers. We therefore modify Goxos so that when a broker replica executes a client request, it introspects the message type. If it is a subscription or unsubscription, the replica will query and update its local subscription table. If it is a publication, the replica will deliver the publication to each of the subscribed clients. Details are given in §3.4.

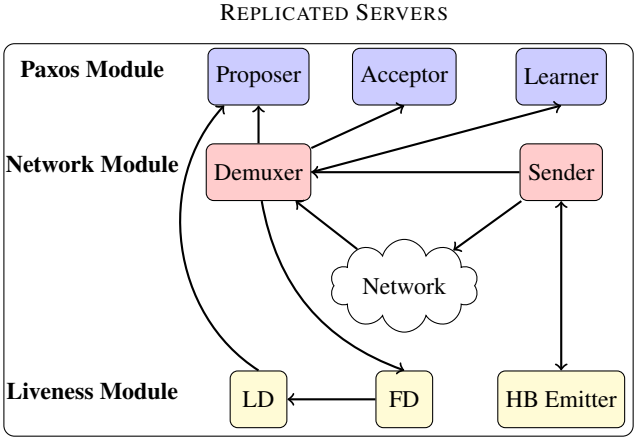


Figure 3: Goxos Architecture [22].

However, we first examine the internals of the Goxos framework. Figure 3 shows the main modules of Goxos, which we organize into three parts: first, the Paxos module, which includes the complete Paxos protocol. Second, the Network module, which handles all networking in Goxos. The Network module contains a Demuxer and a Sender as submodules. The Demuxer handles all incoming connections and relays received messages to the appropriate Paxos module at the local replica for further processing. The Sender module is responsible for sending messages to other repli-

cas per request from other Goxos modules. These two modules, taken together, emulate remote channels between Goxos agents. Finally, the Liveness module, which handles the failure detection and leader election necessary for Paxos. The three different modules communicate with each other through Go's channels. In the figure, a single-ended arrow pointing from a source module to a destination module signifies that the source can send a message to the destination over a one-way channel. A double-ended arrow signifies that both modules can send and receive to one another over a two-way channel. For example, the Demuxer module sends messages to the proposer, acceptor, and learner (which are in the Paxos module). Since Paxos itself must be able to handle many concurrent activities, the Liveness module, Network module, and Paxos module are all implemented as concurrently executing goroutines.

As a base framework for building fault-tolerant services, Goxos offers sophisticated user interfaces for higher level applications to invoke. Listing 1 shows four main interfaces available to application developers.

Listing 1: Goxos interface.

```
type Handler interface {
  Execute(req []byte) (resp []byte)
  GetState(slotMarker uint) (sm uint, state []byte)
  SetState(state []byte) error
}

func NewGoxosReplica(uint, uint, string, app.Handler) *Goxos

func Dial() (*Conn, error)

func (c *Conn) SendRequest(req []byte) ([]byte, error)
```

Server applications can create a replicated service with the `Goxos.NewGoxosReplica` function. This will construct a new replica. The first two arguments of `NewGoxosReplica` are the id of the replica and the id of the application. The third argument is a string describing the application. Finally, the last argument is a type that implements the `app.Handler` interface. The `app.Handler` interface must be implemented by an application that uses the replication library. This interface defines several methods that must be implemented on the type: `Execute`, `GetState` and `SetState`. The first method, `Execute`, takes a byte slice, which should be a command that can be executed in the application. The `Execute` method also returns a response from the application in the form of a byte slice. The second and third methods, `GetState` and `SetState`, are necessary to incorporate a new replica into the system, after the running replicas have made changes to their state, after the initial state.

The client library for Goxos is used to connect to the Paxos replicas, as well as to send and receive responses. The client connection can be created with the `Dial` method in the library. This method returns a `Conn`, representing a connection to the whole replicated service. All of the work of handshaking with the servers and identifying the leader is abstracted away. The most useful method on a `Conn` is `SendRequest`, which can be used to send requests to the replicated service, i.e. the group of servers. The client request is a byte slice, meaning that if the application wants to send Go structs or other complex data types as commands to the replicated service, it first must marshal them into byte form. Similarly, the return value is also a byte slice, which represents the response from the service. Note also that a client must wait for a response from the Goxos servers, or more precisely the leader, before it can send the next request.

3.2 System Architecture and API

P2S, as a fault-tolerant pub/sub service, is comprised of a client library, a replicated server cluster with the Goxos library as the core, and a client handler deployed at servers. The client handler reside at the servers and receives messages from client applications (publishers and subscribers). The client library is used by clients to communicate between the client handler at the servers. The replicated server cluster handles all incoming client requests via the client handler, and orders requests to achieve total order, even in the presence of failures. Finally, the server application executes the ordered client requests. Figure 4 shows the P2S architecture.

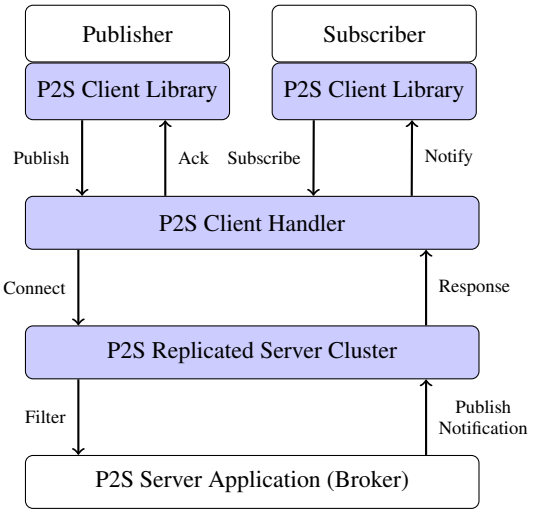


Figure 4: P2S System Architecture.

The **P2S client library** offers standard pub/sub style applications a set of client APIs. The client library communicates with servers, sends out client requests (which can be publications, subscriptions, or unsubscriptions), and receives corresponding responses for the client application to interpret. As shown in Listing 2, the library defines a pair of data structures that applications must use, two standard interfaces, and several methods.

Listing 2: P2S Client Library.

```
type Request struct {
  Ct      CommandType
  Cid     string
  Topic   string
  Content string
}

type Response struct {
  ToType  CommandType
  Ack     string
  Topic   string
  Content string
  Subs    []string
}

type PublicationManager interface {
  Publish(topic, content string)
}

type SubscriptionManager interface {
  Subscribe(topic string) chan []string
  Unsubscribe(topic string)
}

func PDial(account string) PublicationManager
func SDial(account string) SubscriptionManager
func (sm *submngr) awaitPublications(notifyChan chan []string)
```

`Request` and `Response` define the data format that client applications must use. `Ct` in `Request` and `ToType` in `Response` represent the command type, which can be 'Publish', 'Subscribe', or 'Unsubscribe'. `Cid` in `Request` denotes the client ID, which is used by servers as a key to identify the corresponding client connection. `Topic` and `Content` represent publications and subscriptions. Lastly, `Subs` in `Response` is an array of subscribers' ID that is filtered by the servers for publication delivery.

The interface `PublicationManager` is implemented by a publisher's application. `Publish` calls are used by the application to issue a publication. `Publish` takes two arguments as input: the topic and content of the publication. Similarly, the interface `SubscriptionManager` is implemented by a subscriber's application. This interface has two methods, `Subscribe` and `Unsubscribe`, both taking a string of topic as an argument. The `Subscribe` returns a channel on which string slices can be sent. This channel is used by the `awaitPublications` method, which is used by a subscriber to wait for publications on a topic for which it has previously subscribed through the `Subscribe` method.

Both `PDail` and `SDial` are called when an application initiates. They return instances of `PublicationManager` and `SubscriptionManager`, respectively, that the application later invokes.

The **P2S client handler** is initiated on server startup. The client handler is the frontend of the replicated server cluster that handles client connections. It receives connection attempts from clients, stores client requests (a publication or subscription), passes the request to the backend P2S server application for filtering, and receives the processed result, and finally sends back the response to relevant clients. The processed result may either be an acknowledgement to a publisher or a publication for which there are matching subscribers. Listing 3 shows the set of functions in the client handler library.

Listing 3: P2S Client Handler.

```
func (ch *ClientHandler) greetClient(conn net.Conn)
func (ch *ClientHandler) handleRequest(req *Request)
func (ch *ClientHandler) handleResponse(resp *Response)
```

The `greetClient` function starts up an infinite loop waiting for potential client connection attempts. It responds to the `Dial` method that the client calls; it identifies the client address and ID, then stores the client connection object in a local connection pool.

The `handleRequest` function receives client requests, checks each request to see if it has been executed before, generates a response for new request, and stores both the request and response.

The `handleResponse` function is called immediately after a response is generated by the `handleRequest` method. `handleResponse` first loops over the client connection pool, identifies the client that sent the request, then pushes back the response to the client. The `handleResponse` function then introspects the request type. If the request is a publication, `handleResponse` initiates the filtering, finds the subscribers that are interested in the topic in the client connection pool, and delivers the publication to all the subscribers.

Finally, the **P2S replicated server cluster** is the service with our modified Goxos framework as the core. It does not differentiate between different client message types. It simply treats each client message as a Paxos proposal and runs it through the consensus protocol. It then passes the client message to backend server application for interpretation.

3.3 ZapViewers Application

In order to evaluate the capabilities of P2S, we built a fault tolerant TV viewer statistics application based on an existing centralized (non-replicated) pub/sub system [23, 24] deployed at a real IPTV operator. We refer to this as our ZapViewers application. In our evaluation, we use recorded event logs from the real deployment.

A high-level architecture of our ZapViewers application is shown in Figure 5. The application consists of three parts: event publishers (set-top boxes), subscribers (clients interested in viewership statistics), and a replicated broker. A P2S event publisher simulates a fraction (around 180,000) of IPTV set-top boxes (STBs) deployed at customer homes receiving IPTV over a multicast stream. Each STB records viewers' TV channel change information, and sends the event to the IPTV operator's server. The publisher accomplishes this simply by calling our `Publish()` method. Based on these events, the broker computes the TV viewership.

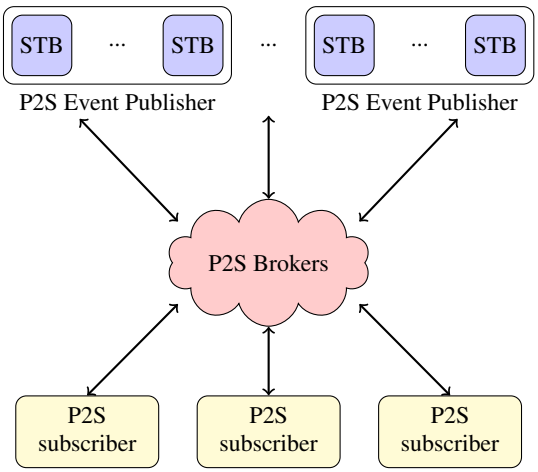


Figure 5: ZapViewers Application Architecture.

A P2S subscriber can either be television broadcasters or commercial entities interested in TV viewership statistics. Such a subscriber is usually concerned about ratings of TV channels, and viewers' channel change behavior. The subscriber that we implemented informs the server of its interested topics, such as top-*N* most viewed TV channels or viewership of some specific channels. The broker then notifies each subscriber of the corresponding statistics. The subscriber calls the `Subscribe()` method to inform the brokers of their interest.

P2S brokers are replicated server applications that appears as a fault-tolerant broker to external event publishers and subscribers. P2S brokers rely on the Goxos framework as their core by implementing system APIs such as the `Handler` interface as described in previous sections. The brokers implement several functions to collect events and computes statistics, including the two shown in Listing 4.

Listing 4: ZapViewers application interface.

```
func numViewers(channel string) int
func computeTopList(n int) []*zl.ChannelViewers
```

Function call `numViewers(channel string)` takes a `channel` name as input from a P2S subscriber and returns that channel's viewership information. Function call `computeTopList(n int)` returns a list of the `n` most viewed channels at a particular instant to the subscriber.

The P2S publisher can generate two event types as follows:
⟨DATE, TIME, STB-IP, TOCH, FROMCH⟩
⟨DATE, TIME, STB-IP, STATUS⟩

DATE and TIME mark the date and timestamp that the event is triggered. STB-IP is the IPv4 address of the sending STB unit. TOCH and FROMCH indicate the new channel and the previous channel that the STB unit is tuned in on. STATUS is a change in status of the STB, which is either volume change on a scale of 0–100, mute/unmute, or power on/off. The event is encoded in text format, and its size is typically less than 60 bytes.

Events have either 4 or 5 fields. An event with 5 fields represents a TV channel change event, and such an event does not contain STATUS. An event with 4 fields contains a STATUS in the 4th field, but does not have the fields TOCH or FROMCH.

3.4 Broker Algorithm

The core of our P2S application is the replicated service provider, the broker. A broker does a handful of back-end jobs, including maintaining subscriptions, storing P2S events as publications, filtering and matching, and delivering publications to subscribers. We explain the essentials of the broker algorithm in the following.

Algorithm 1 Broker Algorithm

```
1:  Initialization:
2:  ST                                          {Subscription Table}
3:  ReqChan                                     {Request Channel}
4:  RespChan                                    {Response Channel}
5:  PropChan                                    {Proposer Channel}
6:  R                                           {Reply Queue}
7:  Paxos                                       {Paxos Variant}
8:  P                                           {Message Type: Publication}
9:  S                                           {Message Type: Subscription}
10: leader                                      {Current leader}
11: on event req ← ReqChan                      {Monitor Request Channel}
12:     handleRequest(req)
13: on event resp ← RespChan                    {Monitor Response Channel}
14:     handleResponse(resp)
15: on event prop ← PropChan                    {Monitor Proposer Channel}
16:     executePaxos(prop)
17: on event executePaxos(prop)                 {Execute Through Paxos}
18:     RespChan ← genResp(prop)
19:     if prop.Type == S then
20:         update(ST)                          {Update Subscription Table}
21: on event handleRequest(req)
22:     if myid == leader or allowDirect[Paxos] then
23:         if req is new then
24:             PropChan ← req                  {Send into Paxos Module}
25:         else ack(R.find(req))               {Re-reply old Request}
26:     else redirect(req)                      {Redirect to Leader}
27: on event handleResponse(resp)
28:     R.add(resp)
29:     ack(resp)                               {Acknowledgement}
30:     if resp.Type == P then                  {Invoke Publication Delivery}
31:         C = filter(ST)                      {Filter and Match}
32:         deliver(C, resp)                    {Deliver Publication}
```

A brokers maintains the following variables: the subscription table ST, the channel for subscription and publication requests *ReqChan*, the channel for acknowledgements and to-be-delivered publications *RespChan*, the channel for sending proposals to Paxos *PropChan*, the queue of replies $\mathcal{R}$, the Paxos variant in use *Paxos*, and two message types for introspection **Publication** and **Subscription**.

When a broker starts up, it initializes several routines: monitoring the request channel *ReqChan*, the response channel *RespChan*, and the proposer channel *PropChan*. When a broker receives a new client request, it invokes the handleRequest(req) method. The handleRequest(req) function call first checks if itself is the current Paxos leader. If not, it checks whether the Paxos variant in use permits direct message routing between non-leader replicas and the client. Fulfilling either of the two conditions means that the request is handled immediately. Otherwise, the broker redirects the request to the Paxos leader.

The broker checks if the request is a new one. If so, it sends the request on the proposer channel *PropChan*, triggering a run of the Paxos algorithm. If it is an old request, it simply finds the response in the reply queue by $\mathcal{R}.find(req)$, and ack() the client once more.

When a request is sent on the proposer channel, the broker invokes executePaxos($prop$) and the request is passed through Paxos. The execution result generated by genResp($prop$) is sent into the response channel *RespChan* immediately. In addition, the broker introspects the message type and if it is a subscription, the broker updates the subscription table ST.

On detecting a new response from channel *RespChan*, the broker calls handleResponse($resp$). The broker adds the response to the reply queue $\mathcal{R}$, and ack($resp$) back to the client. This means the broker introspects the message type and if it is a publication, the broker traverse the client connection pool, filters out the subscribers by examining the subscription table using filter(ST), and finally delivers the publication to all subscribers on the topic.

Each valid client request is executed through the whole cycle and the broker is capable of executing multiple concurrent requests. This is enabled by the Paxos variant in use. Our Goxos framework provides Multi Paxos [25], Batch Paxos [25] and Fast Paxos [26] for the time being. In our P2S application, we use Multi Paxos with $\alpha = 10$ concurrent Paxos instances. We further describe the evaluation in §4.

4. EVALUATIONS

In this section, we evaluate both our ZapViewers application with different replication degrees and the original non-replicated version. We evaluate end-to-end latency, throughput, and scalability under different settings.

4.1 Experiment Setup

All experiments are carried out in our computing cluster composed of GNU/Linux CentOS 6.3 machines connected via Gigabit Ethernet. Each machine is equipped with a quad-core 2.13GHz Intel Xeon E5606 processor with 16GB RAM.

For our experiments, we obtained recorded event logs from a real commercial IPTV provider. The experiments are carried out using 1, 3, 5, and 7 broker replicas. The experiments using only 1 broker are our baseline, as they represent the non-replicated ZapViewers application. The experiments using 3–7 broker replicas allows our system to tolerate 1–3 crash failures. We use up to 24 event publishers, with each event publisher simulating 180,000 STBs, and a small number of subscribers. In the real deployment, each STB caches local channel changes for channels with retention longer than 3 seconds. These cached events are sent to the server every 10 seconds. Indeed, the number of the event publishers (STBs) is typically large, while the number of the IPTV viewership statistic subscribers (e.g., TV broadcasters and other commercial entities) is relatively small. However, while the event volume produced by each STB is relatively low, the aggregate becomes significant.

In all experiments, we use pipelined Multi Paxos [25] with $\alpha = 10$. That is, ten distinct Paxos instances can be decided concurrently. Even though they are decided concurrently, their processing takes place sequentially. Each Paxos instance comprises a batch of STB events to be processed by the broker replicas in sequence.

4.2 End-to-End Latency

We first assess the end-to-end latency. Herein, we define end-to-end latency as the duration between the sending of an event and the corresponding receive at an active subscriber. The latter is inferred from the notification corresponding to the source event. For calculating end-to-end latency, we record a timestamp when a publication is issued by a publisher, and this timestamp is kept by brokers in the execution result that is delivered to any subscriber. The subscriber is therefore able to calculate the latency by comparing the original publisher's timestamp and local time.

Figure 6 shows the latency of our ZapViewers application in different configurations, namely non-replicated, with 3, 5, and 7 replicas, each tolerating 0, 1, 2, and 3 crash failures, respectively. We observe an increase of end-to-end latency in all four experiments as we increase the number of P2S event publishers. We vary the number of publishers from 1 to 24.

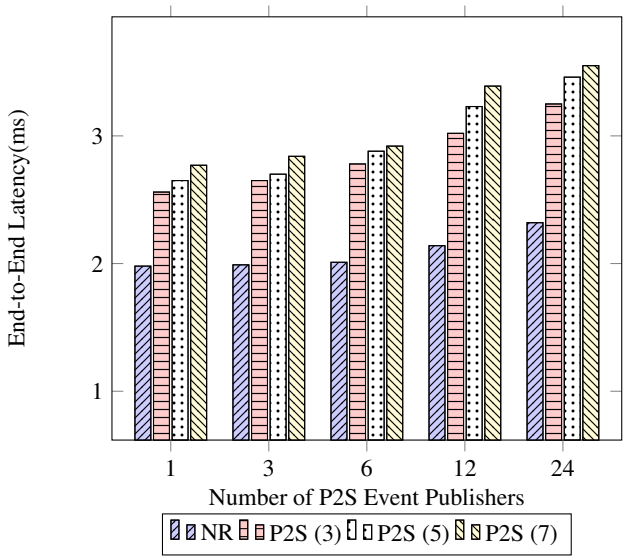


Figure 6: End-to-end latency for various numbers of publishers

The latency of the original non-replicated ZapViewers application varies from 1.98 ms under light load up to 2.32 ms under high load. As expected, all experiments with our replicated ZapViewers implementation show higher latencies than the non-replicated version. That is, we observe an overhead of 0.58 ms (29%) under light load, and 1.23 ms (49%) under high load. Still, from our subscribers' point of view, this latency overhead is barely noticeable.

Also as expected, the latency gradually increases as the number of publishers increases. Since we pipeline events using the Goxos library, the latency increase is small. For the non-replicated broker, the latency overhead of accommodating 24 publishers instead of just 1 corresponds to 0.34 ms (17%). In comparison, with 3, 5, and 7 brokers, latencies are 0.69 ms (26%), 0.81 ms (30%), and 0.78 ms (28%) higher when the number of concurrent P2S event publishers grows from 1 to 24.

We also see that higher replication degrees (indicated by the different bars in Figure 6), imposes only marginal latency overhead.

4.3 Broker Throughput

We assess the broker throughput for the same configurations as in our latency evaluation, as shown in Figure 7. We define broker throughput as the number of publications that are processed by the broker per second. We run experiments in a pipelined manner, with ten distinct instances decided concurrently.

We first observe that for small workloads, all experiments achieve almost identical throughput. With fewer than 6 publishers, the throughput reduction is less than 6% between non-replicated broker and the 7-replica broker.

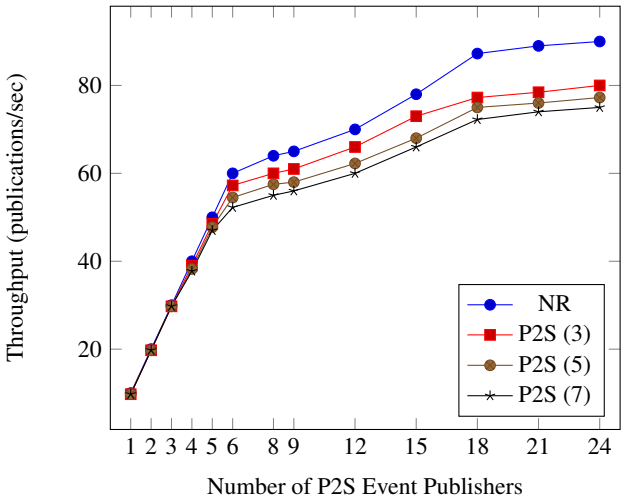


Figure 7: Broker throughput for varying number of publishers.

When the number of publishers is higher than 5, the non-replicated application achieves slightly higher throughput than its replicated counterparts. The throughput drops as little as 4.58% compared to the non-replicated application. As shown in Figure 7, the peak throughput of the original non-replicated application, when there are 24 publishers, is 90.00 publications per second. In comparison, the peak throughput with 3, 5, and 7 replicas are 80.04, 77.25, and 75.03 publications per second, which are 9.96%, 14.16%, and 16.63% lower than non-replicated service, respectively.

Higher replication degree results in consistently lower throughput. Similarly to latency, the overhead caused by this is 6.5% on average. This is explained by the fact that in Paxos, higher replication degree does not cause significant performance degradation.

4.4 Scalability

We evaluate the scalability of our ZapViewers application by varying both replication degrees and the number of event publishers.

Table 1 presents the latency and throughput degradation of the ZapViewers application as the replication degree varies. We compare each instance with a counterpart that has one replication degree lower. As shown in the table, the non-replicated application outperforms all replicated counterparts. With only 1 publisher, the latency of the non-replicated application is 29.9% higher than that of P2S (3). With 24 event publishers, it is 40.08% higher. However, latency drop becomes less noticeable as the replication degree increases. For instance, with 1 publisher, latency of P2S (5) is 3.51% lower than that of P2S (3). With 24 event publishers, it is only 6.46% lower.

Throughput decreases slower on the other hand. When the workload is fairly low, with fewer than 3 event publishers, the difference is barely detectable. The non-replicated application is 11.11%

Table 1: Latency (upper table) and throughput (lower table) drop of ZapViewers, compared to the counterpart that has one replication degree lower. #p is the number of publishers.

	$\#p=1$	$\#p=3$	$\#p=6$	$\#p=12$	$\#p=24$
P2S (3)	29.29%	33.1%	38.30%	41.12%	40.08%
P2S (5)	3.51%	1.88%	3.59%	6.95%	6.46%
P2S (7)	4.52%	5.18%	1.38%	4.95%	2.60%
P2S (3)	2.50%	1.25%	4.58%	5.71%	11.11%
P2S (5)	0.00%	0.00%	4.80%	5.68%	3.43%
P2S (7)	0.00%	0.00%	4.12%	3.61%	2.91%

Table 2: Latency drop (upper table) and throughput rise (lower table) of ZapViewers, compared with its own performance when p differs. Values with parenthesis in red represent positive improvement. The number of publishers is denoted by #p.

	$\#p1-3$	$\#p3-6$	$\#p6-12$	$\#p12-24$
NR	0.50%	1.00%	6.46%	8.41%
P2S (3)	3.51%	4.90%	8.63%	7.61%
P2S (5)	1.89%	6.66%	12.15%	7.12%
P2S (7)	2.52%	2.81%	16.09%	4.71%
NR	(200.00%)	(100.00%)	(16.66%)	(28.57%)
P2S (3)	(205.12%)	(92.43%)	(15.28%)	(21.21%)
P2S (5)	(205.12%)	(83.19%)	(14.22%)	(24.09%)
P2S (7)	(205.12%)	(75.63%)	(14.83%)	(25.00%)

higher than P2S (3). With higher replication degree, throughput varies between 2.91% and 3.43%.

We also compare the performance change for replication degree when the number of P2S event publishers varies, as shown in Table 2. For each application, latency rises with more event publishers. With high replication degree, the latency gradually becomes stable, approaching the peak latency when the number of P2S event publishers is more than 12. When the number of event publishers is greater, the latency decreases much slower, thereafter.

This trend is consistent with the improvement of throughput when the number of event publishers differs. As shown in the table, under low workload, throughput improves almost linearly. When there are more than 6 event publishers, the increase becomes gradually slower. For instance, from 6–12 event publishers, P2S (7) throughput grows 14.83%, or 2.47% per publisher. Also from 12–24 event publishers, growth is 25%, or 2.08% per publisher. This indicates the brokers have almost the maximum processing rate.

To summarize, P2S scales very well when the replication degree and the number of event publishers increases. This demonstrates that our system can retain its efficiency even when we build a system that can tolerate more failures.

5. RELATED WORK

The topic of constructing reliable pub/sub systems has been widely studied [5–13]. By using periodic subscription [5], subscribers actively re-issue their subscriptions. By flooding the messages, this can prevent message loss and ensure subscribers eventually receive all the publications to their subscriptions. On the other hand, through event retransmission [6, 7], brokers exchange acknowledgment messages to ensure that the corresponding messages are delivered. Both periodic subscription and event retransmission work well in preventing message loss instead of handling broker/link failures. In order to guarantee that messages are correctly delivered in the presence of broker/link failures, several publications have proposed redundant paths [6, 8, 9, 13], where the overlay topology includes redundant paths to ensure that at least one path between the corresponding publisher and subscriber is correct. For instance, Gryphon [12] uses virtual brokers, where each broker maps to one or more physical brokers, such that at least one broker is correct and forwards the messages along the path. Indeed, the most straightforward way to use redundant paths is to replicate every broker. However, this may consume high bandwidth and become very inefficient in the absence of failures. Furthermore, prior work in this area usually ensures that messages or events are delivered, where the order of events are not considered.

There has been considerable work in developing total order algorithms [27, 28]. A class of algorithms arranges brokers into groups and uses interactions between groups to compute message order [29]. This type of solution works well under static topology since group membership knowledge can be difficult to maintain in dynamic networks. On the other hand, it is natural to use a single sequencer or several decentralized sequencers [30, 31] to handle message order. A single sequencer is easier to maintain but is a single point of failure. In contrast, decentralized sequencers are more resilient to failure but require every message to be routed to a certain sequencer. This imposes topology constraints and can be less efficient.

Several efforts [9, 10] exploit the topology overlay in pub/sub systems to achieve certain ordering properties in the presence of broker/link failures. Kazemzadeh et al. [9] use a tree-based topology and achieve per-publisher total order by having each broker forward redundant messages to several brokers. A stronger pairwise total order is achieved by Zhang et al. [10], where the intersecting broker of different paths resolves the possible conflicts of message order. However, this has a more complex algorithm to handle broker failures and is less efficient in the presence of failures. Recent work [32] has also used similar tree-based overlay approach to tolerate a configurable number of arbitrary malicious failures in any part of the pub/sub system, with small divergence from traditional pub/sub specifications and forwarding schemes. This work achieves BFT based on replicated state machines using authenticated broadcast and reliable broadcast instead of Paxos. It assumes an initial structure of tree overlay and ensures FIFO order. However, it is not extended to achieve total order of publications. In comparison, P2S takes the simplest yet effective topology and algorithm to achieve pairwise total ordering in the presence of failures. In addition, the flexibility of the framework and our fault tolerance library make it easy to adapt to more scalable systems.

Fault tolerance techniques for highly available stream processing usually consider that no data is dropped or duplicated [33–36]. Most of them assume a failover model and require $f+1$ replicas to mask up to f simultaneous failures. Similar to some of the pub/sub approaches, replicated replicas ensure that at least one correct replica continues processing. When an upstream replica fails, the downstream replica switches to another correct upstream replica. Since at least one correct path exists between the source and destination, the data stream can be delivered. SGuard [35] uses replicated file systems to achieve fault tolerance. Each data chunk is replicated on multiple nodes. The data sent by a client is spread to all replicated nodes so that at least one piece is available. It also relies on a single fault-tolerant coordinator using rollback recovery.

6. CONCLUSION

This paper presents P2S, a simple fault-tolerant pub/sub solution that replicates brokers in a centralized pub/sub architecture. Our solution fits naturally in many industrial settings that need certain resilience, without having to rely on complex, overlay networks.

We have shown how our P2S framework adopts traditional fault tolerant protocols to the pub/sub communication paradigm. P2S provides sophisticated generic programming interfaces for higher level pub/sub application builders, and is built upon our Paxos-based, fault-tolerant Goxos library. Goxos switches between various Paxos variants according to different fault tolerance requirements. The flexibility and versatility of the P2S framework aims to minimize the effort required for future development of any pub/sub systems with various resilience needs.

Our results, evaluated based on recorded data logs obtained from a real IPTV service provider, indicate that P2S is capable of providing reliability at low cost. With a minimum degree of replication, P2S imposes low performance overhead when compared to the original non-replicated counterpart.

In future work, we aim to experiment with the P2S framework on Byzantine failure models. We believe that there is a need for Byzantine fault tolerance in certain industrial applications, and believe our work can be extended to adapt to BFT as well.

Acknowledgements

Sisi Duan and Sean Peisert's contributions to this research were supported in part by the National Science Foundation under Grant Number CCF-1018871. Ms. Duan's work was also supported in part by a Leiv Eiriksson Mobility Grant from RCN. Haibin Zhang was supported by NSF grants CNS 0904380 and CNS 1228828. Tiancheng Chang and Hein Meling's contributions were supported by the Tidal News project under grant no. 201406 from RCN. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the employers or sponsors of this work.

7. REFERENCES

[1] J. Reumann, "Pub/Sub at Google," *CANOE Summer School*, 2009.

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

[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] Z. Jerzak and C. Fetzer, "Soft state in publish/subscribe," in *DEBS*, 2009, pp. 1–12.

[6] R. Chand and P. Felber, "Xnet: A reliable content-based publish/subscribe system," in *SRDS*, 2004, pp. 264–273.

[7] C. Esposito, D. Cotroneo, and A. S. Gokhale, "Reliable publish/subscribe middleware for time-sensitive internet-scale applications," in *DEBS*, 2009.

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

[9] R. S. Kazemzadeh and R. Vitenberg, "Opportunistic multipath forwarding in content-based publish/subscribe overlays," in *Middleware*, 2012, pp. 249–270.

[10] K. Zhang, V. Muthusamy, and H. Jacobsen, "Total order in content-based publish/subscribe systems," in *ICDCS*, 2012.

[11] T. Pongthawornkamol, K. Nahrstedt, and G. Wang, "Reliability and timeliness analysis of fault-tolerant distributed publish / subscribe systems," in *ICAC*, 2013.

[12] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *DSN*, 2002, pp. 7–16.

[13] A. C. Snoeren, K. Conley, and D. K. Gifford, "Mesh based content routing using xml," in *SOSP*, 2001, pp. 160–173.

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

[15] S. M. Jothen, "Acropolis: Aggregated Client Request Ordering by Paxos," Master's thesis, University of Stavanger, 2013.

[16] T. E. Lea, "Implementation and Experimental Evaluation of Live Replacement and Reconfiguration," Master's thesis, University of Stavanger, 2013.

[17] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[18] The Go Project. (2013) The go programming language. [Online]. Available: http://golang.org/

[19] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, December 2001.

[20] H. Meling and L. Jehl, "Tutorial Summary: Paxos Explained from Scratch," in *OPODIS*, 2013, pp. 1–10.

[21] 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, Jun. 2003.

[22] S. M. Jothen and T. E. Lea, "Goxos: A paxos implementation in the go programming language," University of Stavanger, Tech. Rep., 2012.

[23] P. Evensen and H. Meling, "A paradigm comparison for collecting TV channel statistics from high-volume channel zap events," in *DEBS*, 2011, pp. 317–326.

[24] ——, "AdScorer: an event-based system for near real-time impact analysis of television advertisements (industry article)," in *DEBS*, 2012, pp. 85–94.

[25] L. Lamport, "Paxos made simple, fast, and byzantine," in *OPODIS*, 2002, pp. 7–9.

[26] ——, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[27] H. Garcia-Molina and A. Spauster, "Ordered and reliable multicast communication," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 242–271, 1991.

[28] K. P. Birman, A. Schiper, and P. Stephenson, "Lightweigt causal and atomic group multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, 1991.

[29] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Trans. Comput. Syst.*, vol. 7, no. 3, pp. 217–246, 1989.

[30] C. Lumezanu, N. Spring, and B. Bhattacharjee, "Decentralized message ordering for publish/subscribe systems," in *Middleware*, 2006, pp. 162–179.

[31] G. A. Wilkin, K. R. Jayaram, P. Eugster, and A. Khetrapal, "Faidecs: Fair decentralized event correlation," in *Middleware*, 2011, pp. 228–248.

[32] L. Jehl and H. Meling, "Towards byzantine fault tolerant publish/subscribe: A state machine approach," in *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, 2013.

[33] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "An empirical study of high availability in stream processing systems," in *Middleware (Companion)*, 2009, p. 23.

[34] J.-H. Hwang, U. Çetintemel, and S. B. Zdonik, "Fast and highly-available stream processing over wide area networks," in *ICDE*, 2008, pp. 804–813.

[35] Y. Kwon, M. Balazinska, and A. G. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *PVLDB*, vol. 1, no. 1, pp. 574–585, 2008.

[36] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer, "Fault injection-based assessment of partial fault tolerance in stream processing applications," in *DEBS*, 2011, pp. 231–242.