REAPS: Quasi-active Fault Tolerance for Big Data Publish-Subscribe Systems

1st Hang Nguyen
Dept. of Computer Science
UC Irvine, USA
hangn5@uci.edu

2nd MYS Uddin
Dept. of CSEE
UM Kansas City, USA
muddin@umkc.edu

3rd Nalini Venkatasubramanian
Dept. of Computer Science
UC Irvine, USA
nalini@ics.uci.edu

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671450

***Abstract*—In this paper, we address the challenges in supporting reliability and scalability in societal-scale notification systems that aim to reach large populations with customized alerts. We explore fault tolerance (FT) techniques in the context of *Big Data Publish-Subscribe systems (BDPS)*, a scalable hierarchical architecture, that meshes big-data platforms (to store and operate on large volumes of data) with a distributed pub/sub broker network (to manage and communicate with a large number of end subscribers). The role of brokers in this architecture is critical since they serve to mediate interactions between subscribers and the backend big data system. We propose the REAPS (*REliable Active Publish Subscribe*) framework that can handle different classes of broker failures including randomized failures and geographically-correlated failures (as in a natural disaster). REAPS implements a low *overhead* fault tolerance service using a *primary-backup* approach; key features include the ability to exploit subscription similarity among brokers and techniques for quasi-active *state replication* to support fast recovery and delivery guarantees of notification services. We implement REAPS and conduct measurement studies on a prototype BDPS platform using real world usecases. We further evaluate REAPS under various failure scenarios to explore the scalability and performance of our proposed FT mechanisms via simulation studies.**

***Index Terms*—Big Data Publish Subscribe Systems, Publish Subscribe Systems, Fault Tolerance**

I. Introduction

Societal scale notification systems have transformed how people request for and receive information today - traffic notifications, extreme weather alerts (NOAA alerts), social media feeds (Twitter) and public health systems (COVID exposure alerts) are examples. At the heart of several such systems is a publish/subscribe based architecture where users subscribe to events of interest proactively and receive notification messages when such events occur [1]–[5]. With the rise of big data platforms and cloud computing technologies, information platforms hosted at a backend big data management system (BDMS) can serve an important role in transforming messaging platforms to scalable, population scale notification systems. Emerging BDPS platforms combine the advantages of popular BDMS with scalable storage and efficient query processing and distributed pub-sub broker networks, that efficiently route notifications at scale to subscribers. In contrast to traditional pub/sub systems, publishers in a BDPS interact with a BDMS, which is responsible for the ingestion and storage of incoming publications persisted as distinct *datasets*. Subscribers in a BDPS subscribe to channels are realized via parameterized functions or queries in a BDMS; users specify parameter values for more targeted notifications. The broker network assumes responsibility for all interactions with end-users (subscribers) who attach to one *broker*, subscribe to events of interest and receive notifications from this broker. Thus, the broker network serves as an intermediary between subscribers and the BDMS backend [6]–[10]. The BDPS paradigm has several advantages. It enables the creation of enriched notifications/reports for subscribers by combining information from publishers with external pre-loaded data sources into BDMS. With the ability to store results from the executions of defined parameterized functions or declarative queries, BDPS systems make notifications persistent. The BDPS approach enables modularization and separates the matching and delivery functionalities of notification systems between the BDMS and pub/sub platforms. While the resource-capable BDMS can support content enrichment, the user-facing distributed pub/sub platform enables a high fanout to subscribers that is geo-distributed.

Reliable notification delivery is of critical importance to a variety of applications including timely and accurate disaster alerts (e.g. mandatory evacuation in wildfires), instant messaging platforms for interactions, public health warnings etc. The BDPS environment is prone to both small and large failures: hardware failures in various computing platforms at the BDMS, broker and subscriber sides, communication network failures and failures of components in the software stack. We specifically focus on failures within the broker network as they can be particularly impactful in a BDPS system - brokers serve as the conduit to connect large number of subscribers/users to the backend BDMS system. Towards this end, we design and develop REAPS, a low-overhead fault tolerant service for BDPS systems. Key contributions include:

- Design of REAPS, a primary-backup based FT service for BDPS that exploits the unique characteristics of the BDPS architecture (Sec II). REAPS uses a phased approach that combines techniques for backup broker selection and quasi-active state replication.
- Formulation of the backup broker selection problem that exploits knowledge of subscription similarity to reduce

backup maintenance overheads (Sec III).

- Design of a quasi-active state replication protocol that executes synchronous subscriber and subscription state replication and asynchronous notification state replication for low communication overheads (Sec IV).
- REAPS validation via prototype implementation and measurement studies; extensive evaluation under a variety of failure modes via simulation (Sec VI).

II. REAPS: Fault Tolerance for BDPS Systems

The unique features of BDPS systems (nature of shared user subscriptions along with persistence capabilities of the BDMS backend) allow us to design a novel low-overhead approach to providing fault tolerance with scalability in such systems.

A. The BDPS Architecture and Execution

The canonical BDPS architecture is hierarchical with a logically centralized BDMS and a network of distributed brokers connected to end subscribers [6]–[9], [11], [12] - see Figure 1. Each end-subscriber is mapped to a primary broker via a broker-assignment step - this primary broker handles communication to/from the subscriber. Scalability is achieved in BDPS systems via *subscription aggregation*, i.e. aggregating identical subscriptions across multiple users that share common interests. For instance, residents of a community subscribe to events in their neighborhood. For this, multiple identical subscriptions from users mapped to a broker (called "frontend" subscriptions) are fused into a single "backend" subscription to be executed at the BDMS. Consequently, a large number of frontend subscriptions from users translate to a reduced number of backend subscriptions at the BDMS. Publications routed to the BDMS generate notifications (results of executing back-end subscriptions); broker nodes retrieve these results from the BDMS, duplicate and share them with all front-end subscribers. We use the terms *notifications* or *subscription results* or *results* interchangeably. This overall flow reduces network and backend overheads and distributes the notification workload across brokers. To further reduce the overhead of the query processing, channels at the backend are executed *periodically* (instead of as continuous queries when notifications arrive); these periodically executing channels are referred to as repetitive channels. The period of a channel (i.e. frequency at which it executes) is specified at channel creation time. At each channel execution, the underlying queries (subscriptions) are run on a set of associated datasets in the BDMS and the generated results are stored in a designated result dataset in the BDMS and corresponding brokers with *matched subscriptions* - subscriptions that have new results, are notified about the new result availability (through *push messages*). Individual brokers then pull the set of result records for each matched subscription and deliver them to the corresponding subscribers. A Broker Coordination Server (BCS) coordinates and mediates interactions between the BDMS, brokers and subscribers. It serves as a public end-point of the broker network for assigning brokers to subscribers; and also handles management issues such as load balancing [11] across brokers or fault tolerance within the broker network.

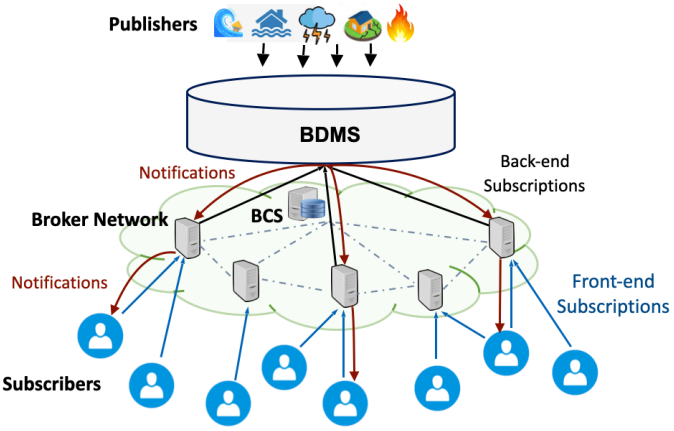


Fig. 1: Big Data Pub/Sub Systems

B. Broker Failure Model and Related Work

Fault tolerance is a concern in any distributed system; it is critical in societal-scale disaster response/alert systems where reliability and timeliness of notifications under extreme events is paramount (e.g. earthquakes, floods) [5], [13]–[15]. Fault tolerance literature has developed a nomenclature for failure models (e.g. crash, link, omission, byzantine failures) and a slew of masking techniques to handle them [14]–[17]. For example, link failures are handled by the introduction of redundant links in the physical network or via application layer multicast/broadcast techniques [14], [15]. State machine replication methods are used to address crash or byzantine failures [16]. Similarly, check-pointing and logging techniques are used for rapid recovery when failures occur. Fault tolerance in pub/sub systems focuses on strategies such as reliable overlay broker network construction, subscription managements and routing protocols for reliable message dissemination [1]–[3]. Techniques to mend and adjust the overlay to cope with broker churn or failure [4], [5], [18]–[20], reconstruct routing states at the recovering broker or buffer messages at the parent of failed brokers for re-transmission [21]. The BDPS architecture does not rely on brokers to communicate with each other by forming overlays to route messages. Hence, these FT approaches for traditional pub/sub are not applicable in BDPS architectures. Therefore, we develop the REAPS service which handles two common types of broker failures, i.e. fault models: i) random individual broker failures due to hardware failures, power outages, software disruptions etc., and ii) geo-correlated failures, where clusters of nearby brokers fail (e.g. in natural disasters).

C. The REAPS Approach

The design and implementation of REAPS exploits unique aspects of the BDPS architecture; specifically, the separation of subscription matching and notification delivery and the persistence of notifications. Each broker serves as the primary or *home broker* for a subset of subscribers; during normal operation, primary brokers manage subscriptions, retrieve and deliver notifications for their attached subscribers. REAPS

leverages the presence of multiple similar servers in the broker network to design a *primary-backup replication* framework for fault tolerance. Traditional primary-backup replication techniques vary in the degree and type of replication [17], [22] including state machine or active replication, semi-active or leader-follower replication [23] and passive replication. We highlight three features in our proposed *quasi-active* REAPS technique to ensure reliable notifications, reduce recovery latency and diminish overhead associated with fault tolerance:

Allocate Backup Brokers: REAPS implements a two-level fault-tolerant framework to cope with two types of broker failures (sect. III) - (a) randomized failures where a few brokers may independently fail randomly and (b) clusters of nearby brokers can fail concurrently due to (infrequent) geo-correlated events such as disasters. Each primary broker will be backed up by two other brokers: one *local backup* in the same cluster to cope with randomized failures and one *remote backup* in a different cluster to cope with geo-correlated failures. A broker may serve as a backup for *multiple* other primary brokers. REAPS will provide a fault tolerant service with no notification loss in case of failures (at some additional cost induced by duplicate notifications).

Exploit Subscription Similarity: REAPS incorporates a subscription-aware approach in selecting and managing backups to minimize the work done by backup brokers. Choosing a backup broker with overlapping subscriptions will incur reduced overheads for state maintenance and thus lead to faster recovery, especially in the local backup scheme.

Support Quasi-active Backup/Replica Management: The REAPS approach aims to create quasi-active backup brokers (active wrt state replication, but passive wrt service replication). Backup brokers are synchronized with the primaries through an adaptive state management protocol. A more aggressive state synchronization technique with higher overheads is implemented at the local backups to handle randomized broker failures which occur at higher probabilities. In detail, local backups other than the similar state synchronization as the remote backups, pre-create and maintain subscriptions to the BDMS on behalf of the primary brokers ahead of failure time to shorten the fail-over process.

REAPS relies on the BCS for collecting system-level information about the broker availability (e.g via heartbeats) and *broker metadata* which includes information about subscribers and thei subscriptions for fault tolerance. The BCS performs the backup broker assignment based on the collected broker states and informs all brokers, which then informs their attached subscribers. The BCS also monitors the broker network for failure detection and triggers the recovery phase when failures happen. During the recovery phase, the BCS informs backup brokers to take over on behalf of any failed primaries. Subscribers from the failed primaries reconnect to their corresponding backups. The BCS may need to run backup reassignment here, since previously assigned backups may have also failed in the meantime. REAPS implements three specialized techniques for i) backup broker assignment; ii) broker state management and replication; and iii) failure detection and recovery. We describe the our techniques in the next three sections.

III. Backup Broker Assignment

Broker networks may suffer from small localized or larger geo-correlated failures. Locality plays an important role in societal pub/sub systems. Brokers that are in close proximity are likely to have many subscriptions in common. For example, users from the same city often have similar interests in receiving notifications on traffic conditions, incoming municipal events, or emergency notifications from the neighborhood schools. However, locality may cause the failure of multiple nearby brokers in the affected regions in disaster events. REAPS includes a comprehensive backup assignment strategy that considers the above issues. We divide the broker network into clusters based on physical locations. The rationale is that clusters that are far away from each other belong to uncorrelated disaster zones so that broker failure in one cluster seldom hampers the broker availability in distant clusters. REAPS assigns each primary broker one local backup among those in the vicinity (same cluster) to leverage subscription similarity, reduce replication cost and one remote backup from a far away location (different cluster) to accommodate geo-correlated failures. REAPS formulates backup assignment as an optimization problem aiming to minimize additional overheads cause by the fault tolerance/replication strategies.

A. Notation

Consider a system with a total of M subscribers, L brokers and N backend subscriptions. In the following, if not otherwise stated, we use the symbols i and j to denote brokers, k to denote a subscription, and u to denote a subscriber. As stated earlier, each subscriber passes its subscriptions to the broker. The broker in turn aggregates the identical subscriptions (the subscriptions that are for the same channel with the same set of parameter values), and passes only the unique subscriptions to the backend BDMS. Let $z_{ik} = 1$ iff broker i has subscription k, otherwise 0, and c_{ik} denote the number of subscribers having subscription k at broker i. Let $e_{ij} = 1$ iff brokers i and j belong to the same cluster, otherwise 0. Thus, REAPS maintains three matrices: $Z = [z_{ik}]$, $C = [c_{ik}]$, and $E = [e_{ij}]$. Given these three matrices, REAPS computes broker assignments that determine local and remote backup brokers for each primary broker in the systems.

B. Broker Assignment Problem Formulation and Algorithms

Let binary decision variables $x_{ij} = 1$ and $y_{ij} = 1$, denote that broker i takes broker j as its local or remote backup, respectively, otherwise 0. Note that $x_{ij} = 1$ only if $e_{ij} = 1$ (both brokers belong to the same cluster) and $y_{ij} = 1$ only if $e_{ij} = 0$ (brokers belong to different clusters). Let the associated decision variable matrices be $X = [x_{ij}]$ and $Y = [y_{ij}]$. The task is to compute these two matrices: X and Y. Each broker is the primary broker for a set of *primary subscribers* and holds their *primary subscriptions*. The primary subscriptions at a broker are *active* and are replicated to the

backups. Replicated subscriptions are called *local backup subscriptions* at the local backup and *remote backup subscriptions* at the remote backup. REAPS keeps local backup subscriptions *quasi-active* and remote backup subscriptions *inactive*. That means a broker *pre-subscribes* non-existing local backup subscriptions to the back-end BDMS and monitors for their notifications, but would not retrieve or deliver to *local backup subscribers*. On the other hand, a broker only stores its remote backup subscriptions without sending them to the BDMS. The remote backup subscriptions are only activated as needed to serve *remote backup subscribers* when failures happen. A local backup subscription is not re-created if it is identical with one of the primary subscriptions at a broker. Similarly, a remote backup subscription is not re-created at recovery if it is identical with one of the primary or local backup subscriptions at a broker. The overhead of the local and remote backup schemes should not be accounted in the same way. More specifically, for the local backup assignment, REAPS aims to minimize the total number of *non-overlapping* local backup subscriptions across all brokers to minimize the *subscription overhead* during normal operation, whereas for the remote backup assignment, it tries to minimize the *largest* number of non-overlapping remote backup subscriptions per broker to minimize the recovery latency. Moreover, in the process of backup assignment, brokers should not be over-committed than its capacity D_i. All these observations lead to the following multi-objective optimization formulation.

The total number of pre-subscribed local backup subscriptions at broker j chosen as a local backup for some other brokers is denoted as:

$$O_j = \sum_{k=1}^{N}(1 - z_{jk}) \left(1 - \prod_{i=1}^{L}(1 - x_{ij}z_{ik})\right) \tag{1}$$

where z_{jk} denotes if broker j has the subscription k, and $\prod_{i=1}^{L}(1 - x_{ij}z_{ik})$ denotes if any broker being locally backed up by j has subscription k or not. The number of non-overlapping remote backup subscriptions incurred by broker j for being the remote backup for other brokers, is given by:

$$R_j = \sum_{k=1}^{N}(1 - z_{jk}) \left(1 - \prod_{i=1}^{L}(1 - y_{ij}z_{ik})\right) \prod_{i=1}^{L}(1 - x_{ij}z_{ik}) \tag{2}$$

R_j takes into account the fact that broker j has already being assigned as the local backup for some brokers and pre-subscribed local backup subscriptions. In the Equation 2, the term $\prod_{i=1}^{L}(1 - y_{ij}z_{ik})$ denotes if any broker being remotely backup by j has subscription k; similarly, the term $\prod_{i=1}^{L}(1 - x_{ij}z_{ik})$ denotes if any broker being locally backup by j has subscription k. The joint local and remote assignment problem is, therefore, to find assignment matrices X and Y, as described in Fig 2.

The constraints for the assignment problem include (i) the local backup broker should remain in the same cluster and the remote backup broker should be from a different cluster (Equation 5), (ii) each broker has exactly one local backup and exactly one remote backup (Equation 6), and

$$\min \sum_{j=1}^{L} O_j \textbf{ and } \min \max_{j=1}^{L} R_j \tag{3}$$

subject to:

$$x_{ij}, y_{ij} \in \{0, 1\}, x_{ii} = 0, y_{ii} = 0, \forall i, j \tag{4}$$

$$x_{ij} \leq e_{ij}, y_{ij} \leq 1 - e_{ij}, \forall i, j \tag{5}$$

$$\sum_{j=1}^{L} x_{ij} = 1, \sum_{j=1}^{L} y_{ij} = 1, \forall i \tag{6}$$

$$P_{j,X,Y} = \sum_{k=1}^{N} \lambda_k c_{jk} + \sum_{i=1}^{L}(x_{ij} + y_{ij}) \sum_{k=1}^{N} \lambda_k c_{ik} < D_j \tag{7}$$

Fig. 2: Joint Local and Remote Assignment Problem

(iii) the maximum workload of any broker, which includes its original workload plus any additional local and remote workload, should be less than its capacity (Equation 7) where λ_k represents the notification data rate for subscription k. The workload is calculated as the sum of the total outgoing volume of data toward the subscribers (regular plus anticipated future workload due to working as the backup for some others). The above optimization problem is NP hard. The problem includes 2 parts: the optimization for the local backup assignment can be reduced to the *generalized assignment problem* [24], [25] and the remote backup assignment can be reduced to the *multi-processor scheduling problem* [26], [27] if these two sub-problems are considered independently. We attempt to solve two sub-problems independently, solving for the local backup assignment using *Least Cost Selection* algorithm to minimize the overall subscription overhead, followed by the remote backup assignment using *Min Max Cost Selection* algorithm which takes into account the pre-subscribed local backup subscriptions from the local backup assignment.

Least Cost Selection (LCS) Algorithm: LCS iterates over the set of brokers and selects a local backup for each broker that produces the minimum additional subscription overhead (Line 10). The algorithm only chooses backup broker candidates that have enough capacity left (Line 8) taking into account the backup assignments made so far. Equation (7) calculates the maximum possible workload of broker j after some assignments being made as X being populated, including the current assignment of broker i to j.

Min Max Cost Selection (MMCS) Algorithm: The MMCS algorithm aims to minimize the maximum number of non-overlapping remote backup subscriptions per broker. MMCS takes into account the local backup assignment in place. MMCS iterates over the set of brokers and selects a remote backup for each broker that produces the minimum of maximum number of remote backup subscriptions which are not covered by primary or local backup subscriptions of one broker (Line 9).

IV. Broker State Management and Replication

This section describes the broker state representation and the replication protocol.

Algorithm 1: Least Cost Selection Algorithm

Find: $X = \{x_{ij}\}$
Initialize: $X = \{x_{ij} = 0, \forall i, j = 1 \dots L\}$
for $i = 1, \dots, L$ **do**
  $overhead = \infty$ /* subscription overhead */
  $b = none$ /* local backup broker selection */
  for $j = 1, \dots, L$ *and* $j \neq i$ *and* $e_{ij} = 1$ **do**

$$P_{j,X_{(x_{ij}=1)}} = \sum_{i=1}^{L} x_{ij} \sum_{k=1}^{N} \lambda_k c_{ik} + \sum_{k=1}^{N} \lambda_k c_{jk} \quad (8)$$

    /* check potential workload for broker j if $x_{ij} = 1$ */
    if $P_{j,X_{(x_{ij}=1)}} < D_j$ **then**
      /* calculate subscription overhead if $x_{ij} = 1$ */

$$\delta_j = \sum_{k=1}^{N} z_{ik}(1 - z_{jk}) \prod_{l=1}^{L} (1 - x_{lj} z_{lk}) \quad (9)$$

      if $\delta_j < overhead$ **then**
        $overhead = \delta_j$
        $b = j$
  $x_{ib} = 1$
return X

Algorithm 2: Min Max Cost Algorithm

Find: $Y = \{y_{ij}\}$
Initialize: $Y = \{y_{ij} = 0, \forall i, j = 1 \dots L\}$
for $i = 1, \dots, L$ **do**
  $overhead = \infty$ /* max number of non-overlapping remote backup subscriptions per broker */
  $b = none$ /* remote backup broker selection */
  for $j = 1, \dots, L$ *and* $j \neq i$ *and* $e_{ij} = 0$ **do**
    /* check potential workload for broker j if $y_{ij} = 1$ */
    if $P_{j,X,Y_{y_{ij}=1}} < D_j$ **then**
      /* check the max # non-overlapping remote subscriptions per broker if $y_{ij} = 1$ */
      if $\max_{l=1}^{L} R_{l,y_{ij}=1} < overhead$ **then**
        $overhead = \max_{l=1}^{L} R_{l,y_{ij}=1}$
        $b = j$
  $y_{ib} = 1$
return Y

A. Broker State Representation

Let l_i and r_i denote the local and remote backups for a broker i. Then, we define $l(i) = \{j | l_j = i\}$ and $r(i) = \{j | r_j = i\}$ as the sets of brokers for which broker i works as a local and remote backup, respectively. Broker i's primary state $\Gamma_i = [U_i, S_i, D_i]$ includes *subscriber state* $U_i = \{m\}$, *subscription state* and *notification state*. The subscription state (indexed by subscriber) $S_i = \{m : \{k, ...\}\}$ and notification state (indexed by subscription) $D_i = \{k : \{(m, t_m^k), ...\}, ...\}$ at broker i are represented using maps of key-value pairs. We denote t_m^k as the timestamp of the newest notification that broker i has sent to subscriber m for the subscription k. Each broker i must maintain its own broker state as well as the backup states for other brokers. We refer to these local and remote backup states l-states Γ_i^l and r-states Γ_i^r. By construction, these synced states are obtained from the $l(i)$ and $r(i)$ brokers and indexed by their respective broker identity. That is, $\Gamma_i^l = \{j : \Gamma_j\}, \forall j \in l(i)$ and $\Gamma_i^r = \{j : \Gamma_j\}, \forall j \in r(i)$. The replication process that we describe below works around updating and maintaining the three states: Γ_i, Γ_i^l, and Γ_i^r, across the brokers.

B. Quasi-active State Replication

Different parts of the Γ_i state are synced (or replicated) with the local and remote backups during operation. In particular, when a subscriber joins/leaves the system, creates a subscription or unsubscribes, the subscriber and subscription state are synced as soon as possible. Upon receiving a sync message on subscription state from the primary, the local backup creates or withdraws backup subscriptions accordingly while the remote backup only updates its backup state. Lastly, the frequent notification state updates are propagated asynchronously to reduce the overall overhead. Recall that t_m^k is updated when the BDMS notifies broker i with new results and the results are retrieved and delivered to the subscribers m that hold subscription k. We use τ^k to denote the latest timestamp of the results generated at the BDMS; such results have not necessarily been retrieved by the brokers, nor delivered to their subscribers. The gap between these two timestamps specifies the volume of *pending results* that must be retrieved from the BDMS by broker i for subscription k to deliver to subscriber m. We use η_m^k to represent the latest timestamp of notifications for subscription k which have been updated to the backup brokers for subscriber m. Then, the gap between t_m^k and η_m^k represent the notifications that are out of sync with the backups. We propose two techniques for notification state replication: (a) periodic replication and (b) threshold-based replication. The replication happens periodically (a) or when the volume of out of sync notifications exceeds a pre-defined threshold. The broker state management and replication is formalized in Algorithm 3. The two RPC functions `l_sync` and `r_sync` are called by primary brokers to sync their states to the local and remote backups, respectively.

V. Failure Model, Detection and Recovery

The BCS is the central coordinator for our fault tolerance approach. It frequently receives metadata updates from the brokers, and thus can recover its state quickly after failures.

A. Failure Detection and Recovery

The BCS implements a failure detector with strong completeness and eventual weak accuracy [28]. To detect failures, our system checks for heartbeat messages from correct brokers. Each subscriber maintains two long lived connections, one with its home broker for normal application notifications and the other with the BCS for broker failure messages. The BCS uses the Tornado framework, a non-blocking network I/O, to maintain up to millions of mostly idle connections with every subscriber. Since our system is asynchronous, the failure detector at the BCS can *suspect* one or multiple broker failures if it does not receive heartbeat messages from those brokers within a *time bound*. In this case, the BCS then sends broker

Algorithm 3: State Replication and Management at Broker i

sync-routine: /* called periodically or threshold-based */
    lsync(i, Γ_i)
    rsync(i, Γ_i)
on_notification(k, τ^k)**:** /* on notifications from BDMS */
    results = **fetch** (k, τ^k)
    for online subscriber m subscribed to k:
        push *(results, m)*
        update D_i: $t_m^k = \tau^k$
on_new_subscriber m:
    lsync(i, Γ_i)
    rsync(i, Γ_i)
on_new_subscription k from a subscriber m:
    If $k \notin D_i$: **subscribe** k /* send k to BDMS */
    lsync(i, Γ_i)
    rsync(i, Γ_i)
on_lsync (j, Γ_j): /* on a local update from broker j */
    subscribe $k1 \in S1$
    un-subscribe $k2 \in S2$
    $\Gamma_i^l[j] = \Gamma_j$ /* update broker state j at local backup */
    /* $S1 = D_j \setminus \{D_i \cup \Gamma_i^l(sub)\}$ */
    /* $S2 = \Gamma_i^l[j][D_j] \setminus D_j \setminus \cup_{p \# j} \Gamma_i^l[p][D_p] \setminus D_i$ */
    /* $\Gamma_i^l(sub) = \cup_{j \in l(i)} D_j$ */
on_rsync(j, Γ_j): /* on a remote update from broker j */
    $\Gamma_i^r[j] = \Gamma_j$

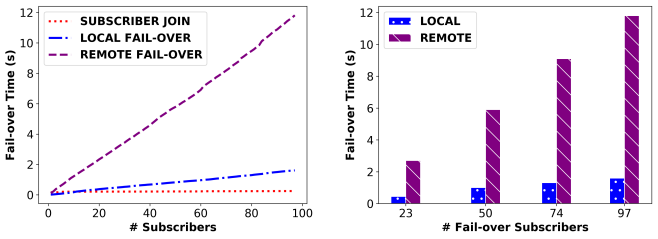
failure notifications to primary subscribers of failed brokers and recommends them to migrate to their corresponding non-suspected local or remote backups. However, if the subscribers can still receive notifications from their suspected home brokers, they can ignore such warning notifications from the BCS. In another scenario, some subscribers may be offline when their home brokers are suspected of failure. As those subscribers come back online, they would connect to the BCS and receive warnings about their home broker failure. If the subscriber is unable to connect to their home broker, then they will migrate to the recommended backups provided by the BCS. To maintain the broker network, the BCS will inform non-suspected local or remote backups to assume the roles of failed primaries. The proposed fault tolerance technique allows recovery from multiple concurrent broker failures as long as the primary broker, local backup and remote backup do not fail simultaneously.

VI. Experimental Evaluation

We evaluated REAPS in a prototype BDPS system and ran simulation studies under different failure models.

A. Prototype Implementation and Measurement Study

We developed a small-scale prototype system containing all the components of a BDPS system. Our BDMS leveraged an existing open-source system AsterixDB [29] with additional features for supporting data ingestion - *data feeds* and data generation - *repetitive channels*. The BDMS was deployed on a cluster of four Intel NUC nodes (i7-5557U 4-core CPU processor, 16GB RAM, 1TB hard drive) and connected via a Gigabit Ethernet switch. We modeled a small broker network of four nodes in two clusters. Each 2-broker cluster was



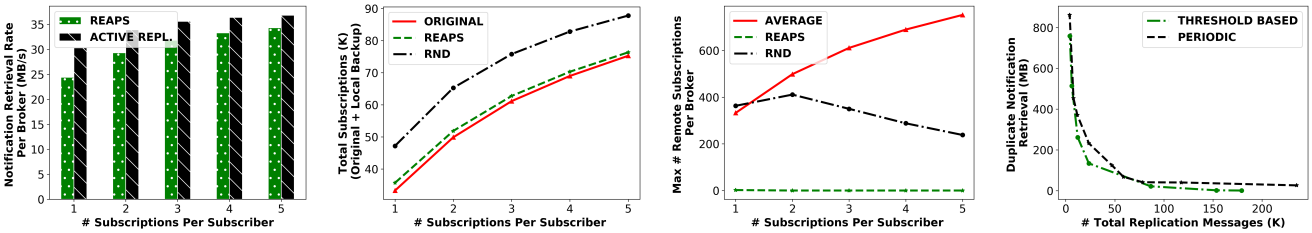
(a) (b)

Fig. 3: Prototype System: (a) Subscriber Migration Time vs. Local Fail-over Time vs. Remote Fail-over Time; (b) Varied # attached Subscribers at the Failed Broker

implemented on Intel NUC servers as described above. The BCS was implemented on another Intel NUC server. Finally, we used a host machine with a Intel Core i5 processor, 8GB RAM and 256GB flash to simulate data publishers and subscribers.

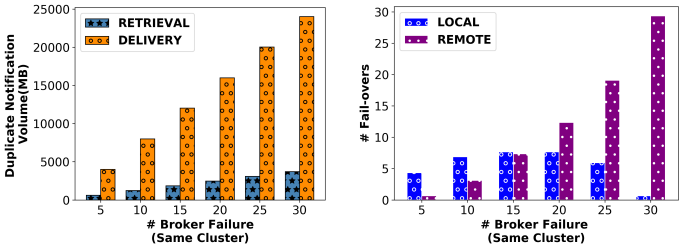
The Notification Application Usecase: We built a hypothetical emergency notification application for the BDPS system whose goal was to quickly notify subscribers of nearby emergency events by considering their current physical location. We defined a repetitive channel *EmergenciesNearMe* which executed every 10s to compute new emergency events that intersected in time and space with subscriber locations. The generated subscription results were delivered to subscribers via their respective brokers. We modeled 400 subscribers that were randomly assigned to brokers. Each subscriber created one distinct subscription to the channel, and passed the subscriber's username as a parameter as follows: *subscribe to EmergenciesNearMe('userName')*. We used the Opportunisitic Network Environment (ONE) [30] simulator to generate two trace files that modeled subscriber movements and emergency events. The application maintained separate datasets in the BDMS cluster: subscriber locations (updated via connections with home brokers) and emergency publications (published into the BDMS via integrated data feed mechanisms). We note that the application can leverage external datasets, such as weather data, traffic routes, and shelter locations, to generate enriched alerts.

Modeling Failures and Measurement Studies: We implemented REAPS in a prototype BDPS system as a proof-of-concept and for real measurement studies. We examined the two local vs. remote fail-over processes caused by a single broker failure. Fig 3a illustrates the progress of the recovery process from the failure of a single broker that served 97 subscribers. The red line shows the progress in the migration of backup subscribers; the blue line shows the progress of the local fail-over process; and the purple line shows the progress of the remote fail-over process. The local fail-over takes significantly less time than the remote fail-over because the remote fail-over needs to create backup subscriptions. Fig 3b demonstrates that the fail-over duration is directly proportional to the number of migrated subscribers (23,50,74,97) in the local/remote fail-over schemes. This result motivated the design of REAPS.



(a) Notification Overhead (b) Subscription Overhead (c) Max # Remote Subscriptions Per Broker (d) Replication Message Overhead

Fig. 4: Single Broker Failure



(a) (b)

Fig. 5: Multi Broker Failure

B. Simulation-Based Evaluation

We further evaluated REAPS's overhead and performance through the simulation studies under various broker failure models. The simulation consisted of 1 BDMS, 100 brokers partitioned into 3 clusters, 100K subscribers, and 100 repetitive channels. The channel execution periods ranged from 10 to 600 seconds. Each channel had 10 distinct subscriptions and the result size per execution for each subscription ranged from 10KB to 10MB. We considered the following types of fault tolerant overheads: i) *subscription overhead* due to the backup broker assignment problem; ii) *replication overhead* due to the state replication mechanism; and iii) *notification overhead* due to the failure recovery.

REAPS vs. Active Replication (AR): We evaluated REAPS against a naïve AR approach where subscribers connected and sent subscriptions to their home brokers as well as their backups. Backup brokers retrieved notifications for backup subscribers but did not deliver them unless the primaries failed. The AR enabled the fastest recovery but produced the largest notification and subscription overheads. On the other hand, REAPS incurred no notification overhead, and only a small subscription overhead for the "quasi-active" backup subscriptions. Fig 4a shows the notification retrieval rate per broker between REAPS vs. AR. We found that exploiting subscription similarity among brokers minimized the notification overhead incurred during AR as well.

Evaluating Backup Broker Assignment Techniques: We compared the incurred subscription overhead between REAPS vs. random (RND) local backup assignment in regard to the total *inherent* subscriptions. The red line in Fig 4b represents the number of original subscriptions across brokers. The green line represents the total original subscriptions and local backup subscription overhead in REAPS. The black line represents the total original subscriptions and local backup subscription overhead using RND. REAPS exploited subscription similarity and hence produced less subscription overhead. Figure 4c shows the maximum number of non-overlapping remote backup subscriptions per broker in REAPS (green line) vs. RND remote backup assignment (black line). The red line represents the average number of original subscriptions per broker. REAPS was optimized to create less non-overlapping remote backup subscriptions per broker for shorter remote fail-over. In this simulation, REAPS produced zero non-overlapping remote backup subscriptions.

Evaluating State Replication Methods: We evaluated the messaging overhead during normal operation and the duplicate notification volume during failure recovery of the two proposed state replication techniques. The *duplicate notification retrieval* is defined as the volume of duplicate notifications that the backup broker retrieved from the back-end for the backup subscribers during fail-over. The *duplicate notification delivery* was defined as the volume of duplicate notifications received across backup subscribers during fail-over. Fig.4d shows that a replication with a higher messaging overhead yields a smaller duplicate notification volume (retrieval) during recovery. The threshold-based technique incurred a slightly smaller duplicate notification volume at the same messaging overhead compared to the periodic replication technique.

Multi Failure Model: Finally, we evaluated REAPS against scenarios where multiple brokers fail. Fig 5a and b show that a larger number of broker failures within the same cluster results in a higher volume of duplicate notifications and a higher number of remote fail-overs which then lead to a higher recovery latency.

VII. Conclusion

In this paper, we designed and developed REAPS to enable FT BDPS systems. REAPS exploits a hierarchical architecture with a Big-Data back-end and edge brokers as well as application characteristics of societal notification systems (geo and socially correlated interests) to address new tradeoffs between reliability, timeliness and scalability of notification systems. Our future work will extend REAPS with methods to capture the importance of critical notifications and implement ways to ensure that they are handled with high priority.

REFERENCES

[1] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *International workshop on networked group communication*. Springer, 2001, pp. 30–43.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 332–383, 2001.

[3] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi, "Meghdoot: content-based publish/subscribe over p2p networks," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2004, pp. 254–273.

[4] C. Chen, R. Vitenberg, and H.-A. Jacobsen, "Omen: Overlay mending for topic-based publish/subscribe systems under churn," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 105–116.

[5] Y. Zhao, K. Kim, and N. Venkatasubramanian, "Dynatops: A dynamic topic-based publish/subscribe architecture," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, 2013, pp. 75–86.

[6] S. Jacobs, X. Wang, M. J. Carey, V. J. Tsotras, and M. Y. S. Uddin, "Bad to the bone: Big active data at its core," *The VLDB Journal*, vol. 29, pp. 1337–1364, 2020.

[7] S. Jacobs, M. Y. S. Uddin, M. Carey, V. Hristidis, V. J. Tsotras, N. Venkatasubramanian, Y. Wu, S. Safir, P. Kaul, X. Wang *et al.*, "A bad demonstration: towards b ig a ctive d ata," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1941–1944, 2017.

[8] S. G. Jacobs, *A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform*. University of California, Riverside, 2018.

[9] M. J. Carey, S. Jacobs, and V. J. Tsotras, "Breaking bad: a data serving vision for big active data," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 181–186.

[10] X. Wang, M. J. Carey, and V. J. Tsotras, "Subscribing to big data at scale," *arXiv preprint arXiv:2009.04611*, 2020.

[11] H. Nguyen, M. Y. S. Uddin, and N. Venkatasubramanian, "Multistage adaptive load balancing for big active data publish subscribe systems," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 43–54.

[12] M. Y. S. Uddin and N. Venkatasubramanian, "Edge caching for enriched notifications delivery in big active data," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 696–705.

[13] H. Jafarpour, S. Mehrotra, and N. Venkatasubramanian, "A fast and robust content-based publish/subscribe architecture," in *2008 Seventh IEEE International Symposium on Network Computing and Applications*. IEEE, 2008, pp. 52–59.

[14] K. Kim, S. Mehrotra, and N. Venkatasubramanian, "Farecast: Fast, reliable application layer multicast for flash dissemination," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2010, pp. 169–190.

[15] M. Deshpande, K. Kim, B. Hore, S. Mehrotra, and N. Venkatasubramanian, "Recrew: A reliable flash-dissemination system," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1432–1446, 2012.

[16] M. J. Amiri, S. Maiyya, D. Agrawal, and A. El Abbadi, "Seemore: A fault-tolerant protocol for hybrid cloud environments," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1345–1356.

[17] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Primary-backup protocols: Lower bounds and optimal implementations," in *Dependable Computing for Critical Applications 3*. Springer, 1993, pp. 321–343.

[18] P. Salehi, C. Doblander, and H.-A. Jacobsen, "Highly-available content-based publish/subscribe via gossiping," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 93–104.

[19] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 41–50.

[20] V. Setty, M. Van Steen, R. Vitenberg, and S. Voulgaris, "Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 271–291.

[21] Y. Yoon, V. Muthusamy, and H.-A. Jacobsen, "Foundations for highly available content-based publish/subscribe overlays," in *2011 31st International Conference on Distributed Computing Systems*. IEEE, 2011, pp. 800–811.

[22] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Optimal primary-backup protocols," in *International Workshop on Distributed Algorithms*. Springer, 1992, pp. 362–378.

[23] P. Barret, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Veríssimo, L. Rodrigues, and N. A. Speirs, "The delta-4 extra performance architecture (xpa)," in *Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. IEEE Computer Society, 1990, pp. 481–482.

[24] G. T. Ross and R. M. Soland, "A branch and bound algorithm for the generalized assignment problem," *Mathematical programming*, vol. 8, no. 1, pp. 91–103, 1975.

[25] I. H. Osman, "Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches," *Operations-Research-Spektrum*, vol. 17, no. 4, pp. 211–225, 1995.

[26] A. Khan, C. L. McCreary, and M. S. Jones, "A comparison of multiprocessor scheduling heuristics," in *1994 Internatonal Conference on Parallel Processing Vol. 2*, vol. 2. IEEE, 1994, pp. 243–250.

[27] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed systems*, vol. 5, no. 2, pp. 113–120, 1994.

[28] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz, "Rollback-recovery without checkpoints in distributed event processing systems," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, 2013, pp. 27–38.

[29] [Online]. Available: http:asterix.ics.uci.edu

[30] A. Keränen, J. Ott, and T. Kärkkäinen, "The ONE Simulator for DTN Protocol Evaluation," in *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. New York, NY, USA: ICST, 2009.