UNIVERSITY OF CALIFORNIA,
IRVINE

Toward Scalable, Reliable and Efficient Big Data Publish Subscribe Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Hang Thi Thu Nguyen

Dissertation Committee:
Professor Nalini Venkatasubramanian , Chair
Professor Marco Levorato
Professor Michael J. Carey
Professor Sharad Mehrotra

2022

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I would like to express my sincerest gratitude to my advisor, Professor Nalini Venkatasubramanian, who offered me a precious opportunity to join the Distributed Systems Middleware research group at the University of California, Irvine. I want to thank her for all her support and guidance throughout the challenging times during my Ph.D. program. I thank her for giving me valuable advice on all my research topics, encouraging me to participate in industry internships. Without her help, I could not complete this thesis.

I would like to thank Professor Michael J. Carey for detailed advice on my last research topic in many weekly meetings. He has given me valuable suggestions on developing and validating the system simulation model and performance evaluation.

Furthermore, I would like to thank my committee members, Professor Michael J. Carey, Professor Marco Levorato and Professor Sharad Mehrotra, for their precious feedback and comments on my research and thesis.

I want to thank my co-author, Professor Yusuf Sarwar Uddin at the University of Missouri-Kansas City, for his excellent guidance and support in formulating my research problems and helping with the implementation of the Big Data Publish-Subscribe Systems.

I wish to thank all my peers in the Distributed Systems Middleware Group and Information System Group for their friendship, valuable feedback, and support, including Andrew Chio, Fangqi Liu, Praveen Venkateswaran, Qing Han, Guoxi Wang, Qiuxi Zhu, Nailah Alhassoun, Elahe Khatibi, and many others..

Most importantly, I would like to thank my family, parents, husband Hung Pham, and sons for their unconditional love, patience, and endless support in my most challenging times. I am grateful and blessed to have them in my life.

VITA

Hang Thi Thu Nguyen

EDUCATION

Doctor of Philosophy in Computer Science — **2022**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Science — **2010**
University of Texas, Dallas — *Dallas, Texas*

Bachelor of Science in Computer Science — **2008**
Hanoi University of Science and Technology — *Hanoi, Vietnam*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2016–2022**
University of California, Irvine — *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant — **2019–2021**
University of California, Irvine — *Irvine, California*

Graduate Reader — **2016–2018**
University of California, Irvine — *Irvine, California*

WORK EXPERIENCE

Software Engineer Intern — **Summer 2021**
Meta Platforms, Inc. — *Menlo Park, California*

Software Engineer Intern — **Summer 2020**
Microsoft Corporation — *Aliso Viejo, California*

Software Engineer Intern — **Summer 2019**
Microsoft Corporation — *Aliso Viejo, California*

System Engineer — **2010-2016**
Joint Stock Commercial Bank for Foreign Trade of Vietnam (Vietcombank) — *Hanoi, Vietnam*

REFEREED CONFERENCE PUBLICATIONS

Multistage Adaptive Load Balancing for Big Active Data Publish Subscribe Systems **Jun 2019**
13th ACM International Conference on Distributed and Event-based Systems

REAPS: Quasi-active Fault Tolerance for Big Data Publish-Subscribe Systems **Dec 2021**
2021 IEEE International Conference on Big Data (Short Paper)

ABSTRACT OF THE DISSERTATION

Toward Scalable, Reliable and Efficient Big Data Publish Subscribe Systems

By

Hang Thi Thu Nguyen

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Nalini Venkatasubramanian , Chair

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 are publish-subscribe-based architectures, where users subscribe to events of interest proactively and receive notification messages when such events occur. Distributed publish-subscribe paradigms leverage distributed broker networks in-place for matching and routing these events to interested subscribers; this helps increase the scale and scope of information dissemination in such systems.

The rise of big data platforms and cloud computing technologies serve an essential role in transforming messaging platforms into societal-scale notification systems. This thesis proposes and designs emerging Big Data Publish-Subscribe (BDPS) systems - scalable hierarchical architectures for the next generation of enriched and customized notification systems. BDPS systems combine i) the advantages of popular Big Data Management Systems (BDMS) with scalable storage, efficient query processing, and massive data ingestion capabilities from heterogeneous publishers and sources, which generates a vast amount of enriched and customized notifications; and ii) distributed publish-subscribe broker networks for scalable delivery of such notifications to interested end-users.

We explore three challenging problems about scalability, resilience, and efficiency of BDPS architectures under dynamic conditions. First, we investigate the problem of potentially skewed load distributions among brokers due to the dynamic nature of the systems. We develop a multistage adaptive load balancing framework for handling dynamically skewed load distributions among brokers, which affects the performance and ability of the systems to disseminate notifications to subscribers efficiently. Next, we address the issue of fault tolerance in the broker network. We propose REAPS (REliable Active Publish-Subscribe). This fault-tolerance framework can handle different classes of broker failures, including randomized failures and geographically-correlated failures (e.g., in a natural disaster), as broker networks are critical parts of BDPS architectures that mediate interactions between subscribers and the backend BDMS. 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. Finally, we develop techniques for prioritizing and scheduling the delivery of important notifications to end-users at the brokers when the systems experience high workloads. Techniques such as load balancing are not applicable to maximize the user benefit and fairness among users, e.g., in situations where large volumes of notifications must be disseminated in a short time. Overall, we explore and develop services for creating scalable, reliable, and efficient BDPS systems that can ingest petabytes of data and generate millions of enriched and customized notifications to reach mega folks in milliseconds.

Chapter 1

The Need for Next-Generation Societal Scale Notification Systems

In this chapter, we first discuss large-scale dissemination systems and their use for societal scale applications. We then argue that a need exists for a new generation of notification systems. Finally, we introduce our proposed paradigms for next-generation notification systems, outline our goals, discuss relevant challenges, and present our contributions in addressing related research problems.

1.1 Notification Systems and Applications

Notification systems have become ubiquitous, appearing in many application domains and serving as essential parts of our daily lives. Notification systems involve software and hardware systems that deliver messages from data sources to groups of recipients. The messages can be disseminated through various media such as outdoor sirens, SMS, MMS, Email [57]. As a result, a wealth of digital information is being generated daily. Several examples of

modern data sources include social media feeds, weather forecasts (NOAA alerts), and IoT systems for monitoring and situational awareness, including traffic notifications, wildfires monitoring, and car crash detection.

Notification systems are typically used for a large variety of applications. They have a wide range of scopes, such as in universities, hospitals, neighborhoods, and communities nationwide. For example, notification systems can be used as an administrative tool to support communication and classroom interaction, send class cancellation notices, remind of assignment due dates. In addition, they can be used to inform community residents of extreme events such as active shooters, building fires, robberies that may affect them [39, 4]. Notification systems are natural parts of every social network, e.g., Twitter, Facebook, and content sharing platforms, e.g., Youtube, Spotify, that notify users of new content, new album releases, new feed updates from their friends/favorite people whom they follow. They are used in hospitals for communicating appropriate information to patients, staff, visitors, emergency services to alert people of adverse events and to enhance situational awareness [26]. Modern vehicles are often equipped with automatic accident detection and notification systems, which depend upon onboard sensors and appropriately notify emergency responders [5]. In alert systems, alerts and warnings are issued to inform individuals and communities who are threatened by emergencies and natural disasters. These include examples such as tornadoes, tsunamis, wildfires, or hazardous chemical spills to help people who have sufficient time and information to take protective actions, such as sheltering in place or evacuating to safety; this reduces the possibility of personal injury, loss of life, property damage [63, 57].

1.2 The Need for a New Generation of Notification Systems

Data dissemination has a wide range of applications - disaster alerting, event notification, and content distribution systems. Data platforms aim to deliver information to a diverse group of typically geo-distributed end-users. However, the design of scalable platforms capable of disseminating *individualized* and *enriched* notifications to end-users in a timely manner are not well studied. In this section, we outline motivation for next-generation notification systems that can meet the needs at a societal scale.

Existing notification systems such as USGS ShakeCast [85], NOAA alerts, traffic notifications, flood the same warning messages to recipients without any customization or enrichment to meet the needs of specific users. However, considering the particular case of disaster notification systems, users need to be provided with *enriched* and *actionable* notifications that best guide and help them act against adverse circumstances. For example, impacted citizens will need to receive notifications that can provide localized situational awareness, e.g., notifications that are enriched information to help people find the nearest shelter locations to escape from an earthquake or find a bus route to get home during a hurricane when the subway system has been shutdown and flooding hinders the traveling by cars.

The need for contextualized and customized notification is also evident in day to day applications that communities use. Different classes of users may have different prioritization or preferences on the information they wish to receive. For example, parents are often concerned about any adverse events in their children's school neighborhoods. Individuals who commute to and from work daily subscribe to traffic conditions and expect to receive notifications about traffic incidents, congestion specific to their routes. People at home may want to receive notifications enriched with pictures and videos to be informed about the current state of their neighborhood or prefer text messages to avoid significant battery drain on their

personal devices and/or running out of their data budget when they are not connected to the Internet.

1.3 Enabling a New Generation of Notification Systems

With large volumes of Big Data generated daily, and the rise of Big Data Management Systems and cloud computing technologies, there is a clear need and advantage for designing notification systems that can ingest huge volumes of data from numerous data sources to generate meaningful and enriched notifications for end-users. A key observation here is the need for incorporating new information in an *active* manner when it becomes available, without end-users having to explicitly send update requests. An enormous amount of digital information is being generated daily through social networks, new sources, mobile applications, IoT devices, etc., and thus, capturing and ingesting these vast amounts of information can benefit numerous end-users who can now gain timely awareness of situations. However, this is challenging due to the scale and speed at which systems must ingest and process information and the scale and speed at which dissemination systems must deliver customized notifications.

In this thesis, we propose a novel architectural approach to design Big Data Publish-Subscribe (BDPS) systems that combines prominent Big Data Management Systems (BDMS) technology for scalable data ingestion, storage, and processing, with scalable geo-distributed Publish-Subscribe (Pub/Sub) systems for scalable data delivery of customized and enriched notifications to a very large number of end-users [42, 44, 16]. The BDPS approach aims to leverage the benefits of both the BDMS and distributed Pub/Sub systems. An exciting aspect of the proposed BDPS approach lies in our ability to create enriched notifications/re-

ports for subscribers by combining information in publications with external data sources. Furthermore, with the ability to store, enrich incoming streams of publications and subsequently leverage them for generating and persisting notifications at the Big Data backend, BDPS supports both push or pull-based approaches for delivering notifications to end-users, e.g., users can access notifications stored in the systems when desired or convenient.

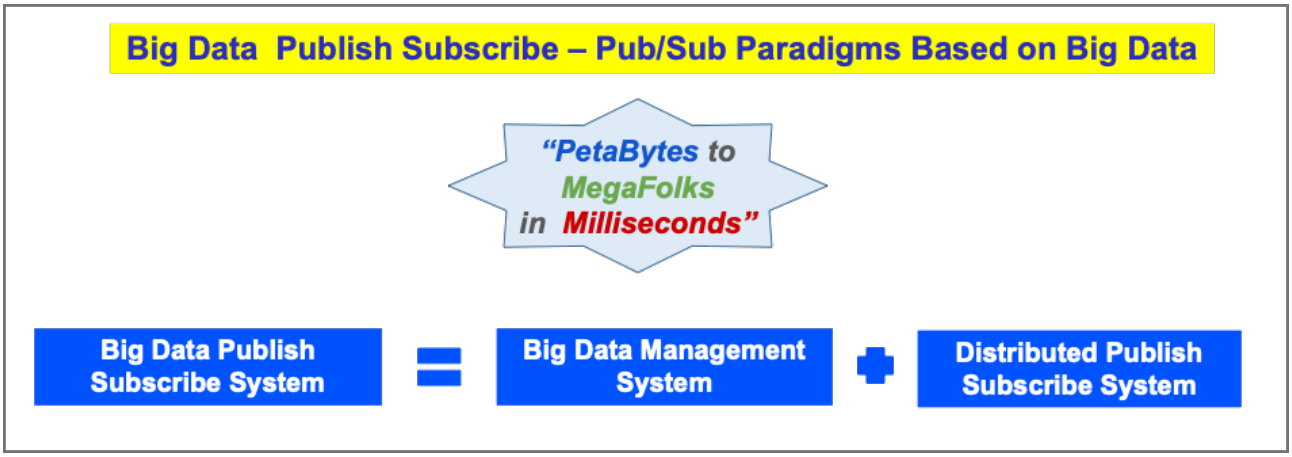


Figure 1.1: Big Data Publish Subscribe Systems

In summary, our goals for designing the new generation notification systems are multi-factored:

- **Produce enriched, individualized, and actionable notifications:** Enriched, actionable, and on-time notifications to help provide end-users with sufficient time and information for taking meaningful actions.
- **Accommodate a large-scale workload of notifications to reach a large population of end-users in real time:** The system should be able to handle dynamic workload spikes of generated notification volume when it occurs. For example, in the case of natural disasters, we may encounter frequent updates and guidelines from agencies that help us make timely decisions. That information must be disseminated to an extensive group of affected communities and individuals. Accurate knowledge of the current situation can help save lives, reduce injuries and property damage.

- **Allow end users to subscribe to the systems and provide active data delivery:** In general, users may not know when notifications and updates arrive from publishers. How does one send requests for information retrieval? Also, systems requiring users to make explicit requests one at a time when there are updates, are slow and do not scale up well. The system should allow users to register subscriptions for interested information ahead of time; this allows the system to generate and deliver new results/updates to end-users when they become available.

1.4 Key Challenges

BDPS systems are distributed systems comprised of various components, including entities outside of the platforms such as data publishers, data consumers, and components within the BDPS platforms, including the BDMS or data cluster in the back-end and the distributed publish-subscribe system in the front-end. Hence, BDPS systems face inherent issues as any general distributed systems, such as scalability, fault tolerance, and recovery. Here, we demonstrate that dynamicity is the key that makes creating scalable, reliable, and efficient BDPS systems a challenging task.

Dynamic Publications, Subscriptions, Subscribers:

First, the subscribers in the system are geo-distributed. The subscriber population may be scattered in one region but crowded in others. Subscribers can leave or join the system or move randomly as desired. Second, the dynamic nature of publications leads to unpredictable notification volume generated at the data back-end. Third, the number and the nature of subscriptions that each subscriber creates can change during operation. All of these factors make the problem of finding an effective subscribers-to-brokers assignment for management and service a non-trivial task. The non-uniform distribution of subscribers and the dynamic

nature of publications from outside publishers and dynamic subscriptions from end-users lead to dynamic notification and subscriber workload incurred by brokers in the system. Designing and building the BDPS system to consider this dynamicity while ensuring the performance of timely notification delivery is challenging.

Dynamic Infrastructure:

The BDPS system is composed of various components where the failure of any component may disrupt the whole system and discontinue the provided service. The failures of the BDMS in the back-end cause failures of critical tasks such as data ingestion and generation. These failures within the broker network disconnect the subscribers from the back-end data cluster. The failures in the communication layer disrupt the data dissemination process. Designing low overhead fault tolerance services, incorporating them into the BDPS systems for fast recovery in the face of failures, and providing a non-disrupted notification service is a challenging task.

Dynamic Notification Value and Dynamic User Preference:

The value of a particular notification to an end-user depends on the delivery time of the notification to the end-user. For example, notifications about the current traffic jam may be valuable for users that have yet to join the congested roads, thus helping them change to alternate routes. However, these same notifications may have less value to users who have already joined such congested roads or do not encounter the congested roads on their routes. Therefore, it is clear that the value of a particular notification for various end-users is varied and depends on user preferences. Furthermore, the value of a particular notification also depends on the nature of the notification or the notification content. Emergency notifications should be more important and urgent than social media notifications. Together with the unpredictable volume and nature of the publications, the unanticipated subscription patterns from end-users and the dynamic nature of notification values and user preferences, the design

of efficient and effective policies and techniques for scheduling notification delivery at the brokers to maximize the value received by all subscribers while ensuring fairness among them, is a non-trivial problem.

1.5 Thesis Contributions and Organization

In this thesis, we address issues in the design and operation of a Big Data Publish Subscribe system to enable the next generation of enriched notification systems that can scale to societal levels.

1.5.1 Thesis Contribution

The thesis consists of three significant challenges that address three main research problems, which are briefly described below.

Multistage Adaptive Load Balancing: The geographically skewed distribution of subscribers and their dynamic interests, combined with the dynamic nature and volume of societal scale publications, may create an uneven load distribution in the distributed broker network. We develop load balancing techniques to mitigate the dynamically skewed load distribution among brokers due to the non-uniform distribution of subscribers, the dynamic nature and volume of publications ingested into the BDMS backend, and the dynamic subscription patterns from end-users which affect system performance and ability to disseminate notifications to subscribers in a timely manner. Our load balancing service comprises techniques for initial end-user placement, smart techniques for subscriber migration that leverage subscription similarity among subscribers, and shuffle technique for subscriber redistribution throughout the entire system.

Fault Tolerant Dissemination for BDPS Systems: 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. The BDPS environment is prone to both small and large failures: hardware failures at various computing platforms at the BDMS, broker or subscriber side, communication network failures, and failures of components in the software stack. We propose REAPS - a fault tolerance service that can handle different classes of broker failures, including randomized failures and geographically-correlated failures (e.g., in a natural disaster). Broker networks are critical parts of the BDPS architectures that serve to mediate interactions between subscribers and the backend BDMS. 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.

Notification Prioritization for Delivery and Fairness in BDPS Systems: The ultimate goal of all notification systems is to deliver valuable, actionable notifications to all target end users in a timely manner. Here, the value of the notifications may be dynamic and depend on several factors such as notification content, time of delivery, and preference from end-users. That is, the value of a particular notification to a specific end-user depends on the nature and content of the notification, the time that it takes for the notification to reach the end-user, and the user's preference for that particular notification relative to other pieces of information. Our goal is to design techniques to quantify value of notifications to end-users and develop techniques for prioritizing and scheduling delivery of important notifications to end-users at the broker level to maximize user benefit and user fairness.

This thesis addresses three main research problems and their given challenges. The thesis contribution is summarized in Figure 1.2

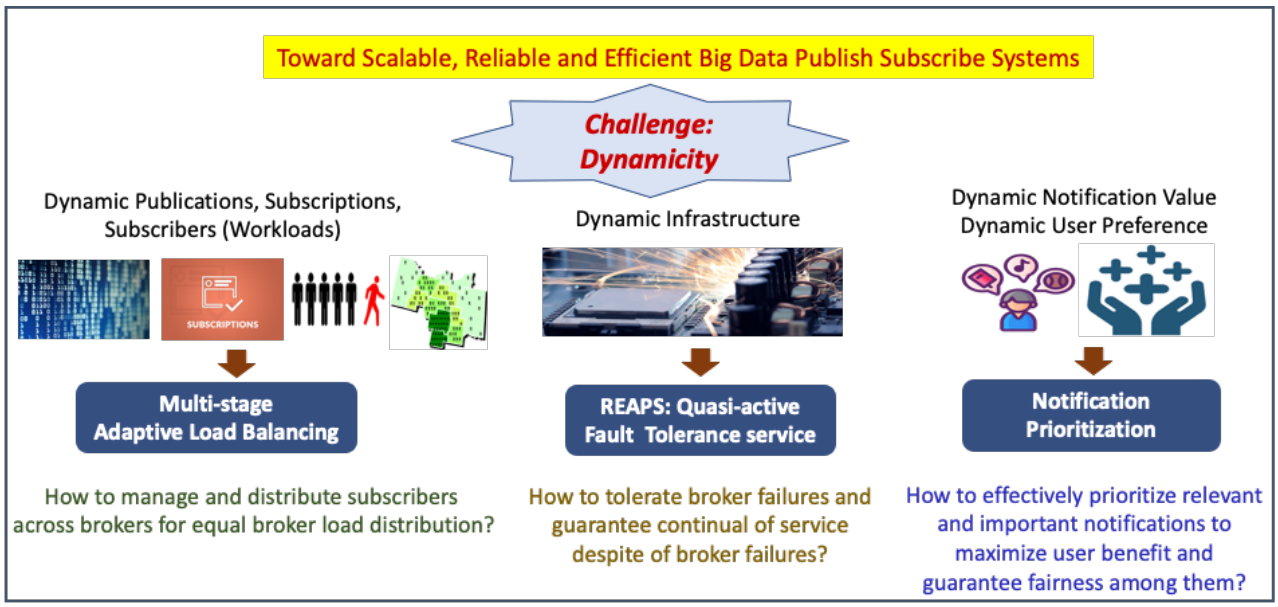


Figure 1.2: Thesis Contribution

1.5.2 Organization of Thesis

In Chapter 2, we survey the existing platforms for notification systems and their limitations.

In Chapter 3, we introduce the design prototype of a canonical BDPS system which will serve as the base system for which the three key challenges of load balancing, fault tolerance, and notification prioritization are developed.

In Chapter 4, we develop a multistage adaptive load balancing service for handling the potential of skewed load distribution among brokers due to the dynamic population and distribution of subscribers among brokers, the dynamic pattern and nature of subscriptions from subscribers, and the dynamic volume and nature of publications ingested into the BDMS. We leverage the feature of subscription similarity among brokers and subscribers to intelligently assign or migrate subscribers among brokers to reduce the overall system workloads, mitigate the dynamically skewed load distribution among brokers for best system performance.

In Chapter 5, we exploit the subscription similarity among brokers and employ a quasi-active state replication approach to design a low overhead primary-backup fault-tolerance service within the broker network to handle failures in the broker network. We provide fast recovery and guarantee continuity of service despite different classes of broker failures, including randomized failures and geo-correlated failures.

Chapter 6 develops notification prioritization techniques for efficient notification delivery when the systems experience unanticipated high workloads. We characterize different importance levels of channels, notifications and develop the model for notification value evaluation which then drives the prioritization policies to maximize the benefit of subscribers and guarantee fairness among them.

Finally, Chapter 7 concludes the thesis contributions and discusses future research directions.

Chapter 2

Existing Data Delivery Platforms and Limitations

Over the years, there has been much effort in designing various communication techniques and leveraging various physical network resources to create scalable, reliable, and efficient data dissemination platforms in academia and industry. In this chapter, we discuss the limitations of existing systems to demonstrate the need for the new proposed BDPS paradigm.

2.1 The Publish Subscribe Systems

We first present the Pub/Sub paradigm, a common framework for exchanging messages in notification systems.

2.1.1 Publish Subscribe System Concepts

The pub/sub systems decouple data senders/publishers and data consumers/subscribers. Publishers publish the messages (events). Subscribers register/sign-up for receiving messages and events of interest via "subscriptions" to the systems. Each entity in a pub/sub system can play the role of a publisher, a subscriber, or both. Subscribers are not directly targeted by a publisher. They are instead indirectly addressed according to the content or topic of the messages. This loose coupling between publishers and subscribers helps seamlessly expand the framework to massive, Internet-scale size and make it suitable for streaming data in real-time [9]. With the rise of cloud-based platforms, typical uses of the pub/sub systems include event messaging, instant messaging, and data streaming. They are commonly used for real-time messaging in many application platforms in big tech companies such as Azure Web Pub/Sub and Google Cloud Pub/Sub [56].

2.1.2 Centralized versus Distributed Publish Subscribe Systems

Generally, publishers and subscribers in a pub/sub system are connected via a broker network. The broker network may compose of a single broker - centralized pub/sub architecture, versus many brokers - a distributed pub/sub system. In a centralized architecture, e.g., IBM WebSphere MQ messaging system (Figure 2.1), TIBCO Rendezvous, all publishers publish messages into a centralized server, and subscribers register their subscriptions to and receive messages of interest from the central server as well. Centralized pub/sub systems may suit the network of small communities or enterprises, while distributed pub/sub systems are required for scalable architectures [32, 31].

Distributed pub/subs require functional layers for scalability, including application-level overlay infrastructure, event routing, and matching algorithms [9]. Broker servers are organized in hierarchical architectures, unstructured and structured peer-to-peer overlay in-

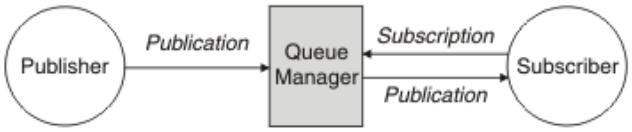


Figure 2.1: IBM WebSphere MQ

frastructures. Each publisher and subscriber attach to one or many brokers in the network. Publishers publish messages, and subscribers send subscriptions to the attached brokers. The broker network matches, routes, and delivers notifications/messages from data publishers to subscribed subscribers [32, 17, 18, 45, 95]. The architectural difference between centralized versus distributed pub/subs is illustrated in Figure 2.2 [31]. Distributed pub/subs can be used in societal scale notification systems to handle thousands of concurrent connections, high volume, and global geographical spread of users.

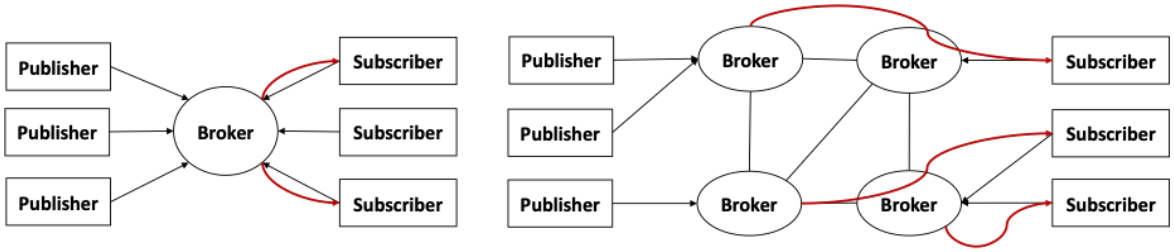


Figure 2.2: Centralized versus Distributed Publish Subscribe Systems

Overall, pub/sub systems can be classified based on several criteria: system topology, subscription model, and routing approach [76]. First, according to the system typology, Pub/Sub systems can be categorized into centralized and distributed Pub/Sub. Second, according to subscription model, pub/sub systems can be categorized as topic-based, content-based and type-based. Third, according to routing approach, pub/sub systems can be categorized into filter-based versus multicast-based approaches.

2.1.3 Subscription Model for Publish Subscribe Systems

Topic-based Model: The earliest subscription model is the topic (channel) based pub/sub. Publishers publish messages to topics (channels) in a topic-based system, and subscribers subscribe to these channels/topics. Brokers deliver all notifications of the same topic to subscribed subscribers. Thanks to its simplicity, techniques such as network multicast facilities or diffusion trees can be used to disseminate messages to interested subscribers [9, 76]. Examples of topic-based pub/subs include: SCRIBE [18], CORBA Notification Service [33], ISIS [13]. The main drawback of a topic-based pub/sub system is its limitation in subscription expressiveness. Hence, a subscriber receives all notifications for the topic it subscribes to.

Content-based Model: Content-based pub/sub systems allow fine-grained subscriptions by enabling subscribers to specify restrictions or predicates on the notification content. The complexity of subscription languages in content-based pub/sub allows flexible subscription expressiveness, but requires complex matching operations for filtering notifications. Brokers match and deliver only notifications that satisfy all constraints specified in subscriptions to subscribers [32, 9]. Examples of content-based pub/subs are Gryphon [77], SIENA [17], PADRES [45]. Group communication, application-layer multicast mechanisms can be used to disseminate messages/events to subscribers [17, 18].

2.1.4 Research in distributed Publish Subscribe Systems

There have been much research efforts in designing pub/sub systems with scalability, reliability, timeliness, and efficiency, including techniques for scalable overlay network, matching algorithms, dissemination protocols, etc [17, 18, 37, 45, 75, 48, 95]. For example, redundant overlay networks, e.g., mesh-based, ring-based structures combined with application-layer multicast, and gossip-based broadcast techniques, have been proposed to handle broker fail-

ures and for fast dissemination [75, 95, 28]. Hierarchical client-server, peer-to-peer, clustering architectures have been introduced for routing subscription advertisements, notifications among brokers for scalability [17, 18, 45]. Multicast tree pruning or repairing for membership management handles subscribers join, leave, or change subscriptions [18]. Techniques such as aggregating subscriptions [19], subscription covering and merging [45, 17, 97] have been studies for scalability. Finally, load balancing, fault tolerance, replication techniques have also been studied in the context of pub/sub systems for availability, scalability, and redundancy [45, 37, 47, 66, 27].

While publish subscribe systems provide active data delivery, they usually support only simple matching algorithms and simple subscription predicates from end-users for scalability. Publications from publish subscribe systems are routed as-is from individual publishers to subscribers, and the ability to generate customized and personalized notifications, therefore, is limited and must be done at the subscriber side.

2.2 Reliable and Timely Notification for Societal Scale Alerting

Typical notification and alerting systems deployed today are best effort both in terms of reliability and timeliness. Mission-critical applications require low latency or real-time dissemination.

One form of information dissemination that arises in distributed, mission-critical applications is called *flash dissemination*. This involves the rapid dissemination of information to a large number of targeted recipients in a time-constrained scenario. Leveraging available resources from end-users to create a large P2P architecture and applying gossip-based protocols is one way to achieve flash dissemination and avoid provisioning dedicated substantial

resources [28]. Exploiting path diversity and the use of multiple data paths in a forest-based application-layer multicast structure can achieve scalability, speed, and reliability as well [53].

Spatial dissemination is another kind of information dissemination where messages are not intended for specific groups of users, but rather recipients attached to specific geographic regions [91]. Spatial dissemination has gained much relevance nowadays in online social networks for targeted marketing and personalized services [59]. With the widespread proliferation of handheld devices, mobile carriers can be connected at anytime, anywhere. Communication technologies enable people to constitute a social network for sharing information whenever and wherever. People tend to constitute a community when they share common interests or are related by family, school, work, transportation or geo-location. Leveraging social relationships, researchers have proposed several solutions for *social aware, geo-social aware* data dissemination [54, 58, 89]. For example, GSFord [54], a geo-social notification system aims to reliably deliver appropriate messages to all relevant recipients under extreme scenarios like disasters, by disseminating the messages to geographically correlated target recipients and socially correlated target recipients.

In temporary circumstances (e.g., entertainment and sporting events, content sharing) or constrained settings (e.g., emergency response, battlefield, wildfires monitoring), temporary network infrastructures such as wireless mesh networks [90], WiFi ad-hoc, cellular networks [92, 29, 30], can be created to enable communication where regular infrastructures are non-existent or to offload the burden on existing network infrastructures. However, flooding approaches typically fail to deliver messages to targeted end-users, while multicast methods fail to individualize messages to distinct end users. Leveraging the prevalence of mobile devices to create delay tolerance or opportunistic networks for data transmission may fail to meet the timeliness requirement [96].

2.3 Other Data Streaming and Delivering Platforms

Content delivery network (CDN) technology has been a popular architecture for low latency content delivery at Internet scale [99]. In CDN, a hierarchy of servers contributes to the delivery of content. The idea behind CDN is to place the content as close to the end-user as possible for low latency. When a user requests content, the request is routed to the closet cache. Suppose the content is not present in the cache storage. In that case, the cache will fetch it from other caches in the same or higher hierarchy level. With the growing demand, numerous CDN at different scales are implemented and deployed worldwide, including Akamai CDN, Amazon Cloudfront, Netfix CDN, Fastly Managed CDN, etc. However, CDN is a *passive* data delivery platform. Caching content is not customized or enriched for an individual end-user.

Similarly, traditional database systems also provide passive data delivery; user requests for information must send explicit queries to systems. In many emerging applications today, queries are evaluated on a database that is being continuously updated, including stock price prediction, sensor monitoring, network monitoring. Active database systems [22, 87] allow users to define trigger functionalities that trigger appropriate actions when monitoring events occur. Continuous query processing systems, on the other hand, allow users to issue queries over a continuous data stream and actively receive new results when they become available [79, 10, 8]. For example, the Tapestry system allows users to issue queries for filtering streams of email, and bulletin-board messages [79]; continuous spatial queries aim to provide updates to the query result as the data objects are moving [68, 64]. However, these systems are not capable of supporting a large number of triggers or complex queries to meet the needs of a vast number of end-users and do not scale to modern data size and arrival rate.

While state-of-the-art data streaming and processing systems which gain importance today

with the rise of cloud-based platforms [36, 78], e.g., Apache Storm [80], Apache Spark [7], Apache Kafka Stream [84], Microsoft Azure Stream Analytics [12], etc. can support processing of real-time events as they arrive to provide timely insights over the never-ending volumes of data with low latency, aggregating data over large windows and processing terabytes of historical data, they are not designed to deliver notifications to a vast number of end-users.

Chapter 3

Big Data Publish Subscribe: Approach and Prototype System

This chapter first introduces our BDPS approach to designing scalable notification systems. We then present a prototype BDPS system developed atop a open-source BDMS, Apache AsterixDB, and demonstrate a hypothetical emergency alert use case which are later used for experimental studies and evaluations for various research problems in BDPS platforms that we study in this thesis.

3.1 The Big Data Publish Subscribe (BDPS) Approach

The proposed BDPS approach combines traditional distributed broker pub/sub platforms with a backend BDMS system to leverage the best of both technologies. As expected, techniques implemented within the BDMS (AsterixDB in our case) provide support for scalable data ingestion, data processing and storage while the broker-based Pub/Sub platform supports scalable data delivery. Consequently, the BDPS architecture is hierarchical with three

layers - (a) a logically centralized BDMS that communicates with (b) a network of distributed brokers that are in turn connected to (c) end users who subscribe to notifications for various types of information. [66, 83, 43, 42, 44, 16] - see Figure 3.1.

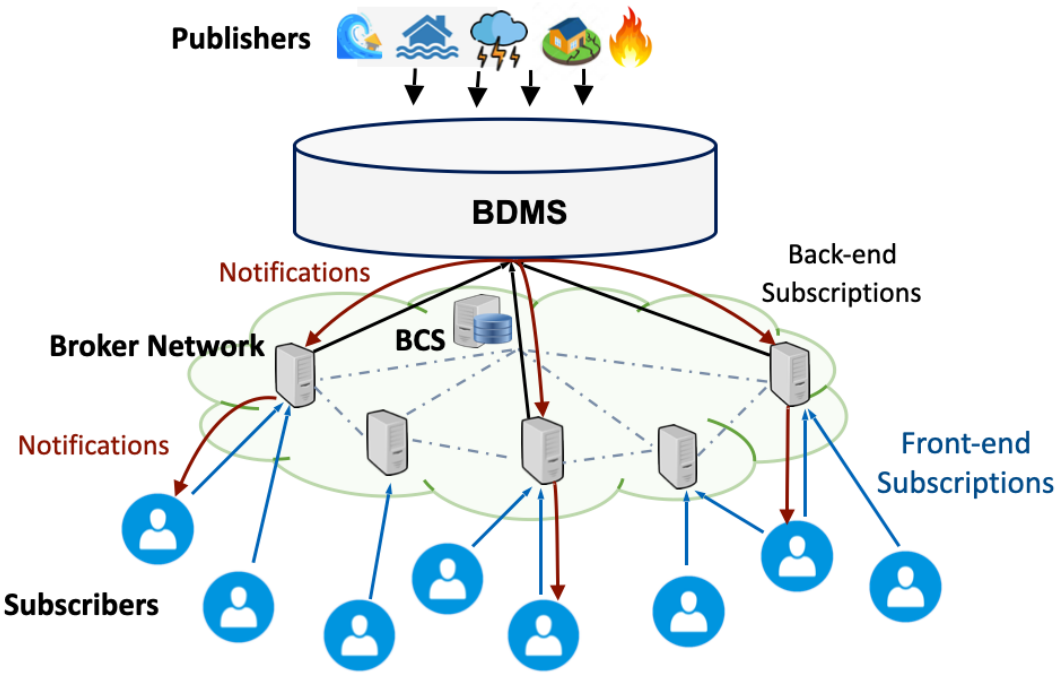


Figure 3.1: Big Data Pub/Sub Systems

An interesting aspect of our proposed BDPS approach lies in the ability to create enriched notifications/reports for subscribers by combining information in publications with external data sources. In contrast to traditional publish/subscribe systems where publications from a single publisher are routed as is to subscribers, publications in a BDPS system are fed into the BDMS backend instead of to a node in the broker network. Additionally, the arriving publications are stored into publication datasets in the BDMS back-end. Each publication dataset stores streams of publications from external publishers. Notifications in BDPS are produced against a set of publication datasets; the information in these datasets is further enriched with selected information in external data sources to create comprehensive reports for subscribers. For example, an location-based extreme weather alert from NOAA(National Oceanic and Atmospheric Administration) may be enriched with a map of current open

shelters and traffic conditions on roads leading to those shelters. Furthermore, with the ability to store incoming streams of publications, match and enrich them at a BDMS, BDPS systems make notifications persistent.

We next describe each component of the hierarchical BDPS architecture in more details.

3.1.1 The BDPS Backend: A Big Data Management System

A key feature of BDPS systems is the presence of a backend BDMS platform (typically a cluster of storage/processing nodes) - this distinguishes the structure and execution of how publications are received and notifications generated as compared to the traditional Pub/Sub paradigm. Here, the BDMS data cluster is responsible for ingestion, storage and management of incoming data (e.g., publications from publishers). In our design, we leverage an existing open-source BDMS, Apache AsterixDB [35] and extend it with features for improved ingest and enrichment. In particular, the BDMS backend implements an abstraction called *data feeds* to allow different data sources (i.e., publishers) to feed their publications into distinct datasets on which queries can be built using a standard SQL-like query language. Instead of subscribing to a topic as is done in topic-based pub-sub systems, in BDPS, subscribers subscribe to functions that can match data from more than one data source (i.e., datasets) and generate notifications accordingly. We refer to these functions as *channels*. Furthermore, these channels can be parameterized to specify custom queries of interest to the user. Datafeeds implemented on the Apache AsterixDB hosted by the data cluster allow subscribers to register their interests by specifying values for those parameters in their subscriptions. Each channel has a qualified unique name and the underlying query is written using the corresponding DML (Data Manipulation Language) supported by AsterixDB.

3.1.2 Data Publishers

Data publishers are the data sources that publish information into designated publication datasets stored in the BDMS. These publication datasets are used to generate notifications based on the stored subscriptions from end users. The extended feature in BDMS, data feeds - also enables users to attach user defined functions (UDF) onto the feed pipeline so that the incoming data can be annotated, enriched if needed before being stored in the BDMS for generating enriched notifications.

3.1.3 Data Subscribers

The end users/subscribers are represented at the lowermost layer. Subscribers are entities that are served in the BDPS systems. BDPS systems aim to deliver enriched and individualized notifications of interest to the end-subscribers in real time. Each subscriber is mapped to a broker via a broker-assignment step. Subscribers register their subscriptions for information of interest to the system via the attached broker. We call them *"frontend" subscriptions*. A subscription is made to a specific channel defined in the BDMS.

3.1.4 Broker Network

The broker network plays an important role in connecting end-subscribers to the data cluster back-end. On the one hand, the broker network receives, manages front-end subscription from end-users, performs (*subscriptions aggregation*) by fusing/aggregating identical subscriptions across multiple users that share common interests. For instance, residents of a community subscribe to events in their neighborhood. Since the number of end-users may potentially be large, managing the large number of subscriptions and end-notifications poses a scalability challenge. To support scalability, we implement novel features within the bro-

ker network. In particular, we introduce the notion of *frontend and backend subscriptions*. Multiple identical *frontend subscriptions* from users mapped to a common broker 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. On the publication side, broker nodes retrieve *notifications/results* from the BDMS, duplicate and share them with all interested front-end subscribers. We use the terms *notifications* or *subscription results* or *results* interchangeably. The overall architecture and the associated techniques reduce network and computation overheads and spread the notification workload across brokers.

To manage the broker network, we introduce a management node called the Broker Coordination Server (BCS) - this is a specialized node in the broker network that is deployed to coordinate and mediate 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 [66] across brokers or fault tolerance [65] within the broker network.

3.2 The prototype BAD BDPS System and Usecase Application

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 publish subscribe platform enables a high fanout to subscribers that are geo-distributed. In this section, we introduce a prototype BDPS platform called BAD. The Big Active Data [3, 2] Project is a multi-university collaborative effort to create a novel active data platform for

scalable data ingestion/processing and scalable distribution of enriched notifications. Architecturally, the BAD BPDS platform consists of two main parts: the BDMS cluster for ingesting, storing publications from external data sources, and generating subscription results for subscribers; and the distributed broker network for managing subscriptions from end-users, relaying them to the backend BDMS and retrieving and delivering results to subscribers.

3.2.1 The prototype backend BDMS - BAD-Asterix

The backend BDMS chosen for the BDPS prototype leverages the open-source big data management system, Apache AsterixDB [1] which supports the scalable storage, searching, and analysis of mass quantities of semi-structured data. The AsterixDB team developed an active toolkit on top of the Apache AsterixDB BDMS - for details, see the PhD dissertation work of Steven Jacobs [44]. We refer to this new extended version of AsterixDB as BAD-Asterix. The active toolkit contains novel concepts and techniques to allow active interactions with other components of the BDPS platform , including:

- **Data Feeds**: to allow scalable data ingestion from external publishers into the BAD BDPS.
- **Data Channels**: to enable scalable data processing and result enrichment to match subscriber interests.

Scalability methods in BAD-Asterix include the implementation of the BAD - Repetitive channel and Query (RQ) engine. A repetitive channel is an active, shared function (parameterized query) in AsterixDB with a time period for execution. The query in the channel is compiled once into a deployed job that will be run repetitively based on the defined *channel execution period* to produce individualized results for all subscribers in every period. Our

thesis is based on this initial prototype of BAD-Asterix platform. Xikui Wang's dissertation [86] further introduces new extensions to the data ingestion framework including *dynamic data feed* which supports complex data enrichment and enable adaptiveness to referenced data update during ingestion; and the new BAD Continuous Query (BAD-CQ) service that executes in a batch-continuous manner for providing continuous query semantics and provide incremental update results to subscribers.

3.2.2 The prototype BDPS distributed Broker Network - BAD Brokers

In our prototype BDPS system, BAD Brokers implement modules for (a) managing subscribers and subscriptions for users connected to the specific brokers, (b)relaying these subscriptions in a scalable manner to the BAD-Asterix BDMS backend,(c) retrieving notification results from the backend BDMS and (d) support effective delivery of enriched notification results to subscribers. The BCS node in the broker network discussed earlier mainly implements modules for managing the broker network, mapping subscribers to brokers and supporting other developed services described later in this thesis (Figure 3.2).

The BCS and BAD Brokers are RESTful HTTP servers built on top of the Tornado web framework. We develop a set of RESTful APIs in both BCS and BAD Brokers to support communication among them and with subscribers. Several basic RESTful APIs in the BCS are *registeruser*, *registerbroker*, *requestbroker* and in broker servers are *login*, *subscribe*, *unsubscribe*, *logout*, etc. We further develop other RESTful APIs in BCS and broker network to support load balancing and fault tolerance services as later described in this dissertation.

Figure 3.3 demonstrates the detailed interaction among BDPS components. During the bootstrap phase, each broker first registers itself to the BDMS backend to send subscriptions and receive notifications of available results at the BDMS. A broker next registers itself with

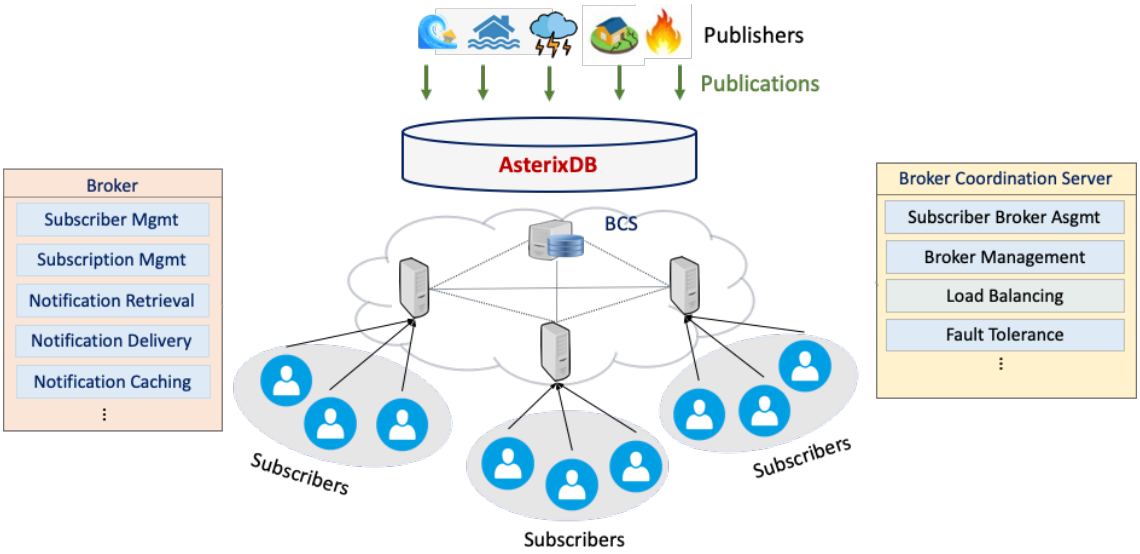


Figure 3.2: BDPS - BAD Broker Architecture

the BCS and it is now available as a prospective broker node that can be assigned to incoming users/subscribers. When a user/subscriber first arrives in the system, it registers itself with the BCS (well-known address) and requests for a primary broker to which it can attach. It then connects to the assigned broker and creates subscriptions of interest over time. The assigned broker node ensures that the instantiated subscriptions are in turn mapped to corresponsing channels at the BAD-Asterix backend. When new results associated with a subscription are generated in BDMS, the associated brokers (those with interested users) are notified. Those brokers then retrieve and deliver results to matched subscribers.

The BAD-broker platform implements a range of techniques to support a scalable subscriber population, handle large numbers of subscriptions from end-subscribers, and enable scalable data delivery of enriched notification results from BDMS to subscribers. An example of one such technique for scalable data delivery was developed in Uddin et al [83] where caching strategies at broker reduce communication overheads associated with data delivery to end-users who may be offline. BAD caching methods determine which result objects to admit into cache and what to drop when the cache becomes full (eviction-based caching). Such caching techniques allow subscribers be able to retrieve results with reduced latency while

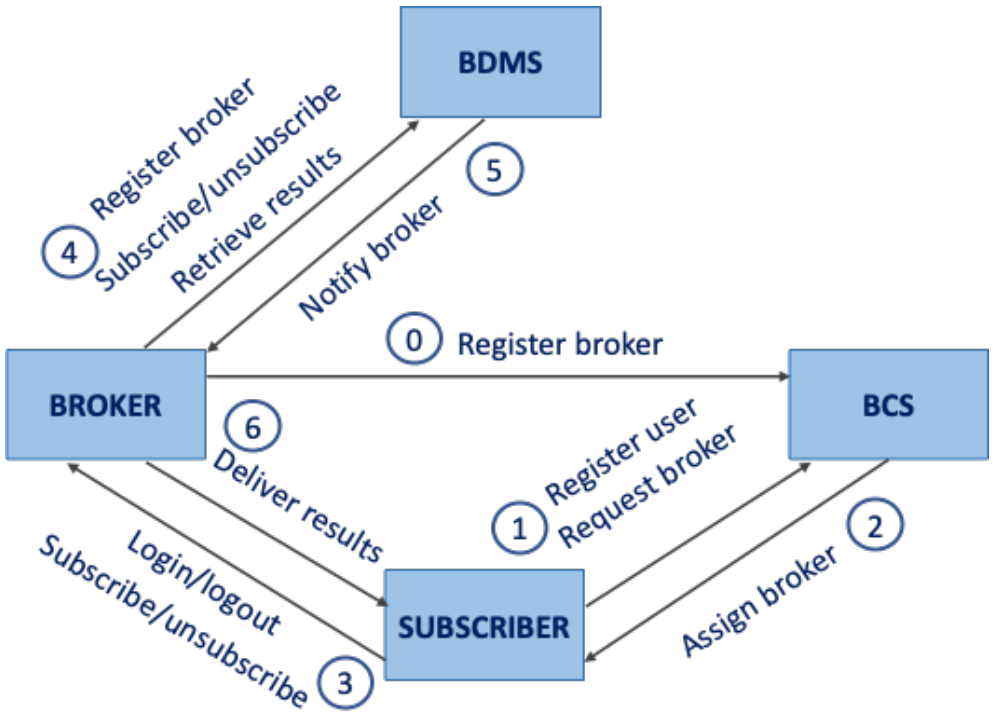


Figure 3.3: Interactions between different BDPS components

also reducing communication overheads. Techniques to enable content adaptation [82] for delivery of rich notifications have also been developed in RichNote, where notifications about the availability of rich multimedia content (e.g. music), can be extended with small samples of the content - such mechanisms are also useful in the context of BDPS systems. The key contribution in RichNote is the ability to enable progressive presentation levels for the content based on the utility of each presentation level to the end-user. RichNote develops techniques to maximize the utility of notifications delivered to end-users under resource budget constraints. This thesis furthers develop techniques and services which enhance the scalability, reliability and efficiency of broker network in supporting, managing and delivering enriched notifications to a large number of end-users.

3.2.3 An Emergency Notification Application

We illustrate below how to leverage BDPS architectures and functions in designing a hypothetical emergency notification application. During an emergency, users wish to receive enriched and actionable notifications beyond the one-size-fits-all general emergency publications. Customized notifications allow end-users to take appropriate protective actions against the adverse circumstances. A specific user could customize locations of interest for emergency events - for instance, a parent would register for information about critical events in neighborhood schools their children attend, citizens will wish to be informed of crises that impact their houses, offices and those of friends/family. What is desirable is that these notifications are enriched with additional information useful in an emergency such as local helplines, shelter locations, escape routes etc.

The emergency notification application created by Big Active Data team (for details, see the PhD dissertation work of Steven Jacobs [44]), defines emergency channels in the BAD-Asterix BDMS to allow end-users to create subscriptions and receive notifications about emergency events happen at their interested locations, or their current locations. The application creates two publication datasets in the BDMS: the *UserLocations* dataset to record the current user location published by each user to the system via the broker network; and the *EmergencyReports* dataset to store emergency events published by various emergency publishers. The BDMS has a pre-existing *EmergencyShelters* dataset that holds locations of about 200 emergency shelters.

The application first needs to create a "emergencyNotifications" dataverse - "data universe" in which to create and manage its data types, dataset, functions and other artifacts.

```
create dataverse emergencyNotifications;
use emergencyNotifications;
```

It then creates publication data types, e.g., EmergencyReport and publication datasets, e.g., EmergencyReports to store publications from individual publishers.

```
create type EmergencyReportType if not exists as open {
recordId: uuid,
severity: int,
impactZone: circle,
timeoffset: float,
timestamp: datetime,
duration: float,
message: string,
emergencyType: string,
userName: string
};
create dataset EmergencyReports(EmergencyReportType) primary key
recordId AUTOGENERATED;
```

The application also needs to define a data feed type for each data publisher, e.g., EmergencyReportFeedType, and create a data feed, e.g., ReportFeed, to ingest publications from the publisher.

```
create type EmergencyReportFeedType if not exists as open {
severity: int,
impactZone: circle,
timeoffset: float,
timestamp: datetime,
duration: float,
message: string,
```

```
emergencyType: string,
userName: string
};
create feed ReportFeed with
{
    "adapter-name": "socket_adapter",
"sockets": "promethium.ics.uci.edu:23231",
"address-type": "IP",
"type-name": "EmergencyReportFeedType",
"format": "adm"
};
```

It then connects data feeds to corresponding publication datasets and instantiates the ingest process.

```
connect feed ReportFeed to dataset EmergencyReports;
start feed ReportFeed;
```

Finally, the application defines query functions, each has a unique name and a set of parameters. The channels are then created using these parameterized functions. Each channel is defined as one query function and a specific execution period.

```
create function recentEmergenciesAtLocation(latitude, longitude){
(select r as reports from
(select value r from EmergencyReports r where r.timestamp >
current_datetime() - day_time_duration("PT20S")) r
where spatial_intersect(r.impactZone,create_point(latitude,longitude)))
};
```

```
create repetitive channel recentEmergenciesAtLocationChannel using
recentEmergenciesAtLocation@2 period duration("PT20S");
```

Subscribers are now able to subscribe to specific channels with provided values of interest for channel parameters. We illustrate below how multiple frontend subscriptions can be coalesced into a single backend subscription below.

Front-end Subscription:

```
user1 subscribe to recentEmergenciesAtLocationChannel(33, -117) on BADBrokerOne;
user2 subscribe to recentEmergenciesAtLocationChannel(33, -117) on BADBrokerOne;
```

Back-end Subscription:

```
subscribe to recentEmergenciesAtLocationChannel(33, -117) on BADBrokerOne;
```

The full Asterix Query Language (AQL) script for creating data types, datasets, index, data channels, data feeds for the emergency application is provided in Appendix A.

Chapter 4

Multistage Adaptive Load Balancing for Big Data Publish Subscribe Systems

In previous chapters, we motivate and introduce the overall design of BDPS systems. We next address the specific aspect of managing dynamic workloads in a BDPS system to enable the next generation notification systems to scale to societal levels. While BDPS systems present desirable features for scalability, ensuring performance and robust operation under dynamic condition is challenging. In this chapter, we present our design and approach for an adaptive load balancing framework to cope with the dynamic skewed load distribution in the broker network.

4.1 Motivation and Overview

Notification systems at the societal scale can induce widely varying workloads based on the applications at hand. A festival event in the neighborhood, an on-going super bowl in the city attract huge interest and subscriptions from the residents or fans (high subscription workload). An emergency application produces a large volume of publications when a disaster event happens, e.g., a tornado or hurricane sweeping through the city. The dynamic user subscription patterns and the dynamic volume and nature of societal scale publications may create load imbalance in the distributed broker network which affect the system performance and ability to disseminate notifications/results to all matched subscribers in a timely manner. The skewed distribution of subscribers geographically may lead to potentially skewed distribution of subscribers among brokers when the system tries to assign subscribers to nearest brokers for locality and low latency.

In order to prevent imbalanced load distribution in the broker network, we first need to determine the factors that contribute to the broker load and the quantification of broker load. The meaningful quantification of broker load must incorporate notions of management loads and communication overheads. The management loads at brokers account for the subscribers and subscriptions management while the communication overheads capture the steps involve in (a) retrieving notifications/results for all subscriptions from the back-end data cluster and (b) disseminating notifications/results from the broker to all matched subscribers. It is worth to note that a broker has the ability to aggregate subscriptions from end users to immensely reduce the end to end overheads and loads.

The dynamic usage pattern of subscribers in term of the number and the nature of subscriptions that each subscriber generates, the unpredictable states of subscribers (active vs. inactive), make the effective assignment of subscribers to brokers a non-trivial task. Load balancing with unpredictable and dynamic workloads in a BDPS system is further influenced

by the subscriber broker mapping strategy and the specific interest of subscribers. An initial formulation of the load balancing problem in such settings is shown to be intractable.

In this chapter, we propose an adaptive load balancing scheme that consists of three phases: i) the *initial placement* phase - which assigns a broker to a subscriber after the subscriber registers itself with the system to start service; policies explored include geo-location based allocation, round robin and random broker selection; ii) the *dynamic subscriber migration* phase - where subscribers are dynamically migrated from highly loaded brokers to lightly loaded ones to handle the fluctuation of broker load distribution during the course of operation; iii) the *shuffle* policy to address the extreme load imbalance condition that re-configures the entire system to optimally redistribute subscribers among brokers.

Chapter Road-map:

- We begin with a discussion of prior related work on a event processing system and its connection to load balancing problem in distributed systems.
- We develop a mathematical model for broker load in BDPS and formulate the associated load balancing problem as an NP hard problem.
- We propose a practical and efficient multistage load balancing framework for BDPS systems (with initial assignment, dynamic migration and shuffle).
- We design novel algorithms for each stage of the load balancing framework in particular, dynamic subscriber migration and shuffle policies to fix load imbalance.
- We implement the proposed techniques in a BDPS prototype system; evaluate the performance of various load balancing policy combinations with real world use cases on the prototype platform and simulation experiments.

Load Balancing and Distributed Event Processing

Load balancing has been a well researched topic since the introduction of parallel and distributed computing and been largely applied in many distributed contexts from distributed databases to high performance computing systems. Sample efforts include key-value pair assignment in distributed networked cache systems [41], request balancing in crowd-sourced CDNs [61], virtual machine assignment in cloud computing [62], object distribution in P2P systems [69], sensor partitioning into clusters in WSNs [38, 60], event key-grouping in complex event processing (CEP) engines [94]. etc. A common thread in load balancing efforts applied across many systems is workload migration [69, 94, 98, 35].

Load balancing in the context of publish/subscribe has been studied in both content-based [23, 37] and topic-based [27] systems using different architectures, where distributed hash table (DHT)-based, tree-based, cluster-based and community-based approaches have been used to organize the broker network. For example, content-based publish/subscribe in [37] uses DHT overlays over a P2P network to route subscriptions and publications to dedicated nodes. Here, load on each peer then is composed of subscription storage load and publication propagation load. Techniques such as zone splitting for subscription storage overload and zone replication for publication propagation overload upon a new peer arrival have been proposed. In cluster-based publish/subscribe [47], event/publication space is organized into partitions. The popularity of the publication determines the number of clusters for each partition - this allows balancing the subscription maintenance load among clusters while publication forwarding load is balanced among brokers in the same cluster. In [23], brokers are partitioned into clusters. Each cluster contains one cluster head who serves only the publishers, and a set of edge brokers who serve subscribers. Clusters are organized into a hierarchical architecture to allow two levels of load balancing: local load balancing among brokers within the cluster and global load balancing among cluster heads of different clusters. Here, bit vectors are used to profile subscription load and offloading algorithms determine an appropriate set of subscriptions for migration to balance multiple performance metrics of a broker including input rate, output rate and matching rate. In community-

based publish/subscribes [88], the author exploits similarity for clustering brokers then uses offloading mechanism within a community for inter-community load balancing and uses filter replication for intra-community load balancing.

Recent work [27] on load balancing for topic-based publish/subscribe system like Apache Kafka also explores strategies to determine a set of partitions for migration from an over-loaded broker to a set of under-loaded brokers where the broker load is characterized by its traffic intensity. Another common way for load distribution in topic-based publish/subscribe is to allocate topics across broker nodes using consistent hashing [34, 71] where each broker node in the network has an unique identifier. The topic is hashed to the same domain as the identifier space and each topic will be assigned to the closest virtual identifier. With efficient placement of broker nodes in the virtual space, each broker node will be responsible for an equal share of topics. Ongoing research on publish/subscribe systems focuses on design choices with regard to overlay topology and routing protocols to improve resource utilization, raise scalability [20, 24], cope with the dynamic of subscriptions and user churn [74, 81, 95] or for efficient message dissemination [48].

4.2 System Model and Problem Formulation

Let us consider a BDPS system with a fixed number of m brokers, $B = \{j : 1, 2, \ldots, m\}$. Let there be a collection of n users in the system $U = \{i : 1, 2, \ldots, n\}$ who create front-end subscriptions that are mapped to a total of q back-end subscriptions. To reduce the overheads associated with instantiating unnecessary notification delivering to inactive/disconnected subscribers. The system state management distinguish between active versus inactive subscribers. Our BDPS platform aims to deliver notifications to online users, that are currently active to receive notifications. Offline users are expected to query the broker for pending notifications when they reconnect. Given the persistent native of results in the BDPS plat-

form, any pending notifications can be delivered to subscribers as they come online. The set of all back-end subscriptions is denoted as $S = \{k : 1, 2, \ldots, q\}$. Note that front-end subscriptions are specific to subscribers and each back-end subscription from a broker maps to a parameterized channel and specified parameter values. A broker can have a back-end subscription held on it only if it has at least one front-end subscription attached to it. In the following, if not otherwise stated, we use i, j and k to denote a subscriber, a broker and a back-end subscription, respectively.

Let y_{ik} denote a binary indicator if subscriber i has a front-end subscription that can be attached to back-end subscription k. Let z_{jk} denote a binary indicator if broker j has to maintain a subscription to back-end subscription k. As we have said, this only happens if the broker has at least one front-end subscription for k originated from some subscriber connected to that broker. Our notion of load balancing is based on the fact that a subscriber is mapped to only one broker. Hence all subscriptions of that subscriber is mapped to that broker. Let x_{ij} denote the current subscriber-to-broker assignment in the system. That is, x_{ij} is set of 1 if subscriber i is attached to broker j. Actually, this assignment metric $X = \{x_{ij}\}$ is the one that the system needs to compute at a certain time. This assignment is done by the BCS.

Now let us consider how the load is constituted at each broker. We define the load of a broker as the sum of total incoming data and the total amount of outgoing data per unit of time. For each back-end subscription at a broker, the broker needs to pull the results from the BDMS each time new results are populated against the associated channel and pushes the same results onto the subscribers who have front-end subscriptions attached to that back-end subscription. Let λ_k be the data rate at which new results are generated for back-end subscription k. Hence, the broker load, denoted as F_j for broker j, has two parts: incoming data volume I_j and outgoing data volume O_j.

The incoming load at broker j can be computed as:

$$I_j = \sum_{k=1}^{q} z_{jk} \times \lambda_k \tag{4.1}$$

where z_{jk} (if broker j has a subscription to k) is given by:

$$z_{jk} = 1 - \prod_{i=1}^{n} (1 - x_{ij} \times y_{ik}) \tag{4.2}$$

Let n_{jk} be the number of total front-end subscriptions attached to a back-end subscription k at broker j, which is given by:

$$n_{jk} = \sum_{i=1}^{n} x_{ij} \times y_{ik} \tag{4.3}$$

Therefore, the total volume of data delivered per unit of time to the attached subscribers from broker j, i.e. the outgoing load, is:

$$O_j = \sum_{k=1}^{q} n_{jk} \times \lambda_k \tag{4.4}$$

Combining the above two terms, we obtain the total load of broker j as:

$$F_j = I_j + O_j = \sum_{k=1}^{q} z_{jk} \left(1 + n_{jk}\right) \lambda_k \tag{4.5}$$

Note that the first element of the load depends on the back-end subscriptions that a broker maintains (which in turn depends on the degree of shared subscriptions by users at the broker, the more the sharing the less the incoming load), whereas the second part is directly

attributed to notification delivery to attached subscribers. This sharing subscriptions among subscribers adds non-trivial complexity and opportunity to the load balancing problem, which is considered in our solution strategies.

We formulate the load balancing problem as a *minmax* problem, that is, the objective of the load balancing problem is to compute the assignment matrix $X = \{x_{ij}\}$ so as to minimize the maximum load across all brokers. One can think of other forms of objective functions to balance load, such as minimizing the difference between the max load and the min load, or minimizing the variance of load, etc. We assume that these choices are orthogonal to the problem at hand and can all be good candidates to check as none of them essentially raises or eases the complexity of the underlying problem. Having said that we define our objective of load balancing is to minimize the maximum load. More formally:

Given:

$R = \{\lambda_k : k = 1, \ldots, q\}$

$Y = \{y_{ik} : i = 1, \ldots, n; k = 1, \ldots, q\}$

Find $X = \{x_{ij} : i = 1, \ldots, n; j = 1, \ldots, m\}$ so as to $\min \max_j F_j$

subject to $\sum_{j=1}^{m} x_{ij} = 1, \forall i = 1, \ldots, n$

The constraint indicates that at any given time each subscriber can be attached to exactly one broker. The above problem is NP-Hard, which can be reduced to the Multi-Processor Scheduling (MPS) problem.

Reduction to the Multi-Processor Scheduling problem. The optimization problem we examine can be seen as an instance of the MPS optimization problem which is a well-known NP-complete problem [25]. In MPS problem, there are a set of m identical machines and n jobs. Each job has a processing time $p_i \geq 0, \forall i \in n$ and the optimization statement

is to assign jobs to machines so as to minimize the maximum processing time among all machines. Our optimization problem can be reduced to the MPS problem where brokers are machines and subscribers are jobs. The MPS problem is a special case of our optimization problem. In more detail, when there is no subscription sharing among subscribers at all in the whole system, says, each subscriber make a unique subscription and each unique subscription has a specific load (specific data rate), then finding the allocation of subscribers among available brokers which generates the most balanced load distribution is exactly the MPS problem.

Since the problem is NP-Hard, we devise algorithms based on greedy heuristics that we describe in the next section. The key idea is to iteratively choose one subscriber at a time (the heaviest one) from from the most loaded broker and assign it to the lightest broker. We observe that when a subscriber is picked for migration, the additional load introduced at the destination broker equals to the sum of λ's of the subscriptions that the subscriber has no matter which destination broker is chosen. On the other hand, the change in incoming load at the destination broker depends on the subscription commonality between the subscriber and the broker itself as the broker only needs to retrieve additional amount of result data for those new subscriptions from the subscriber which are not currently held by the broker. This is the key insight in developing the heuristic algorithms we present in Algorithm 1.

A question remains when to invoke this subscriber to broker re-assignment. Ideally, the re-assignment happens when the system detects an "unbalanced" state from the existing assignment. The BCS keeps track of loads across all brokers and triggers re-assignment when needed (we refer to this as dynamic migration). The BCS uses the coefficient of variation (*cov*), the ratio of the standard deviation to the mean of broker loads, as an indicator of the degree of load imbalance across brokers. The *cov* is calculated as follows:

$$cov = \frac{\sigma}{\mu} \tag{4.6}$$

where

$$\sigma = \sqrt{\frac{\sum_{j=1}^{m}(F_j - \mu)^2}{m}} \tag{4.7}$$

$$\mu = \frac{\sum_{j=1}^{m} F_j}{m} \tag{4.8}$$

A low *cov* value indicates a balanced system whereas the higher value indicates an unbalanced one. The BCS uses a threshold α on *cov* to determine when to trigger the dynamic migration.

4.3 The Multistage Adaptive Load Balancing Approach

Due to the wide range of load fluctuation that each broker may experience over time, we need to constantly monitor the load status of the system to take timely actions in order to balance load in the broker network. In our BDPS platform, such state management is performed by the BCS. Accordingly, all brokers in the system send their load updates to the BCS periodically. This helps the BCS to maintain a global view of the system's overall load across the brokers and to detect if load is skewed across brokers (based on a quantitative metric we define later on). The BCS then implements a multistage approach to balance the load across the brokers in the sense that the system invokes a set of techniques. The first technique is *initial placement* that assigns an incoming subscriber to an existing broker.

Usually, this initial placement may be subscription-agnostic because the subscriber may not have any subscriptions when they join (subscriptions originate later on). Because of the dynamic volume of data generation from the BDMS, the matching rate of each individual subscription, the changing set of subscriptions that each subscriber makes over time, any initial placement can lead to a skewed load across the brokers in the future. Therefore, we propose two more techniques, namely *dynamic migration* and *shuffle* to fix the unbalanced condition depending on how much variation of load among the brokers is detected.

Dynamic migration refers to moving one or more active subscribers from their current brokers to new brokers. The key task here is to decide who moves to where. The BCS takes these decisions, generates *migration plans*, sends them to involved brokers and let them be executed by the brokers. The basic idea is to move heaviest subscribers from heaviest brokers to less loaded brokers and keep doing this until the system reaches a balanced condition. However, when the system experiences an extreme load imbalance, to quickly fix the bad situation and further optimize the distribution of subscribers among the brokers, the system invokes *shuffle*, which redistributes the whole set of current active subscribers over all brokers (*shuffle plan*) (overhauling the entire subscriber-broker assignment from the scratch). While the dynamic migration can be invoked in response to a moderate load imbalance, invoking the shuffle is rare and only happens when the system experiences an extremely unbalanced load distribution. Figure 4.1 describes the implementation model of our load balancing approach.

4.4 The Multistage Adaptive Load Balancing Approach

Our next design and evaluate techniques to address the different sub-problems of our load balancing framework. As mentioned earlier, the three phases are: i) initial placement, ii) dynamic migration, and iii) shuffle. Our overall approach in described in Figure 4.2.

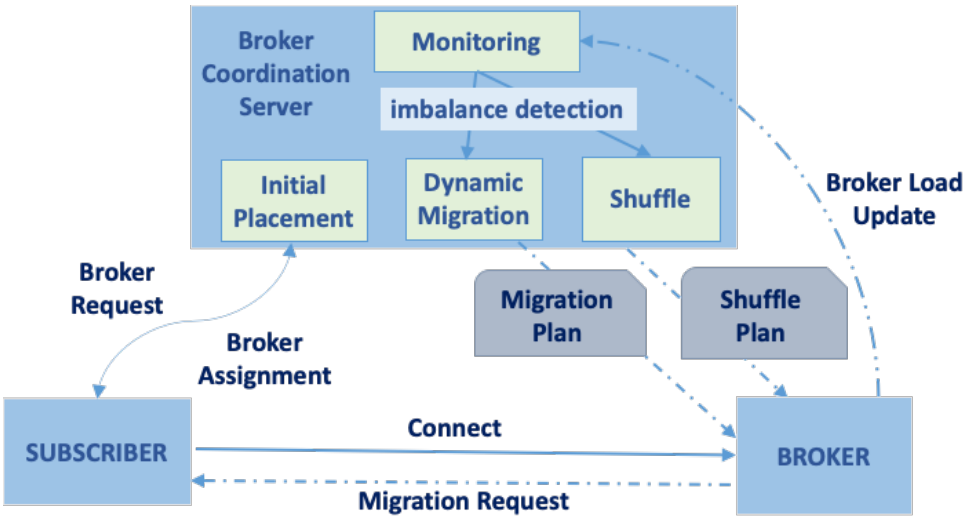


Figure 4.1: Multistage adaptive load balancing implementation model

4.4.1 Stage 1: Initial Placement

The initial placement aims to assign a suitable broker to a newly joined subscriber without knowing apriori the subscriptions of the subscriber. We explore three policies for the initial placement and the mapping of subscribers to brokers.

The Nearest Broker Assignment As the name suggests, when a subscriber sends a request for a broker to the BCS, the subscriber also attaches its current location information to allow BCS return the nearest broker to the subscriber. The distance can be, however, the geographic distance or the network latency (the choice can be implementation dependent). Since the broker network is geo-distributed over a very large area, by assigning the nearest brokers to subscribers the system aims to minimize the latency of results delivery from brokers to subscribers. However, the skewed distribution of subscribers in space may lead to a skewed mapping of subscribers to brokers and can degrade the latency experienced by the subscribers.

The Round Robin Placement The round robin method, on the other hand, balances the number of subscribers assigned to each broker, ideally each broker serves an equal number

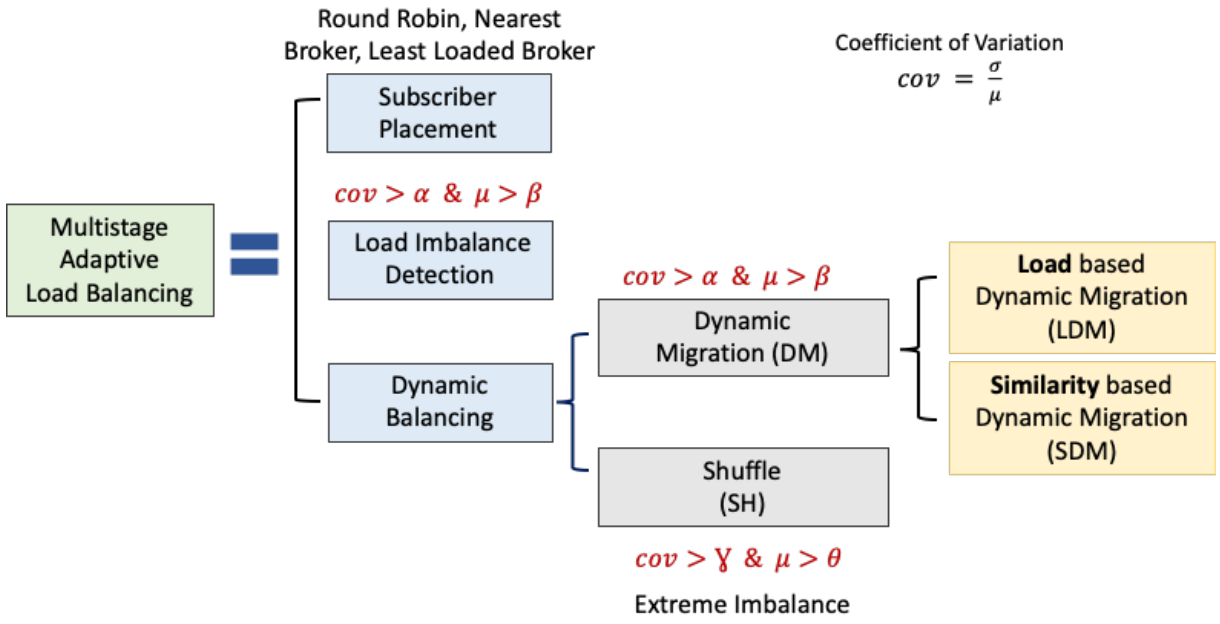


Figure 4.2: Multistage adaptive load balancing approach

of subscribers. Since the load of brokers depends on other factors as well other than only the number of subscribers, such as the number of subscriptions and the nature of those subscriptions, round robin placement also can not guarantee the balanced load distribution.

The Random Placement In this scheme, the BCS assigns an arbitrary broker to each subscriber. If subscribers have a uniform subscription distribution over the subscription space, the random assignment can produce a temporarily balanced system, however fluctuation in publication space may result in imbalance.

One may argue that the least loaded broker placement should be a good policy to explore intuitively. However, in practice, the least loaded broker placement turns out not to perform well, especially when many empty subscribers come to the system at a similar time and they are all assigned to the current least loaded broker which will shortly overwhelm the assigned broker as those subscribers start generating subscriptions.

Algorithm 1: Multistage adaptive load balancing

Input:
$U = \{i : 1, .., n\}$ /* subscriber set */
$B = \{j : 1, .., m\}$ /* broker set */
$R = \{\lambda_k : k = 1..q\}$/* subscription data rate */
$X = \{x_{ij}, i = 1..n, j = 1..m\}$ /* subscriber broker assignment */
$Y = \{y_{ik}, i = 1..n, k = 1..p\}$ /* subscriber subscription matrix */
Output:
$M = \{\}$ /* migration plan */
$scheme \leftarrow \{\text{LDM}, \text{SDM}\}$ /* Load-based Dynamic Migration (LDM), Similarity-based Dynamic Migration (SDM) */
/* extremely skewed load distribution condition */
if $cov > \gamma$ & $\mu > \theta$ **then**
    $M \leftarrow$ SHUFFLE
/* medium skewed load distribution condition */
if $cov > \alpha$ & $\mu > \beta$ **then**
    while $cov > \alpha$ & $\mu > \beta$ **do**
        $b \leftarrow \arg\max_{j \in B} F_j$ /* broker of maximum load */
        $U_b = \{u_i : u_i = \sum_{k=1}^{p} \times y_{ik} \times \lambda_k, \forall i \in U, x_{ib} = 1\}$ /* subscriber load at b */
        for u_i *in* $Sorted(U_b, reverse = True)$ **do**
            $simTable = \{sim_j : sim_j = \sum_{y_{ik}=1, z_{jk}=1} \lambda_k, j \in B\}$ /* similarity between subscriber i and broker j */
            if $scheme == LDM$ **then**
                $b' \leftarrow \arg\min_j F_j$
            if $scheme == SDM$ **then**
                $b' \leftarrow \arg\max_{j, F_j < \mu} simTable$
            if $F_{b', x_{ib'}} = 1 \leq F_{b, x_{ib}=1}$ **then**
                $M \leftarrow \{b : [i, b']\}$
                break
        Update $F_b, F_{b'}$
        $x_{ib} \leftarrow 0$, $x_{ib'} \leftarrow 1$
return M

4.4.2 Stage 2: Dynamic Migration

The initial placement of subscribers to brokers does not guarantee a balanced system over time. Therefore, we design a second phase to readjust the system configuration in our load balancing framework whenever a skewed load distribution is detected. Recall that the BCS

collects current loads of all brokers periodically; it calculates the coefficient of variation *cov* as a measure of load imbalance across the brokers. If *cov* exceeds a threshold α and the average load across all brokers, μ is greater than a lower bound θ (Algorithm 1), the BCS generates a migration plan and sends the plan to all target brokers to initiate the migrations. The parameters α and θ are determined by empirical studies. Leveraging the idea of the *longest processing time* greedy algorithm to solve the MPS problem, we develop two strategies for migration: (a) load based dynamic migration and (b) similarity based dynamic migration as described in detail below. Our general intuition is to keep selecting a subscriber from the most loaded broker in each iteration as a candidate to migrate to a less loaded broker. By doing that, after every iteration, the maximum load of brokers is decreased or the number of brokers with maximum load (if there are more than one brokers with maximum load) is reduced until no further subscriber migration can be performed or the system reaches the balanced state.

Load-based Dynamic Subscriber Migration: In this technique, the subscribers from the heaviest broker are ranked by their individual load. Then, each subscriber is checked for a valid migration in the order of decreasing load. A migration is said to be valid if after migration, the load of the destination broker does not exceed the original load of the source broker. For the load-based migration scheme, the destination broker is always the currently least loaded broker. For each valid migration, one entry is added to the migration plan that contains a list of tuples specifying which subscribers need to migrate from their current brokers to new brokers. The list is indexed by the source brokers and at the end of the algorithm (once the plan is populated), the respective brokers are notified with the list of subscribers to shred off and the destination brokers for those subscribers.

Similarity-based Dynamic Subscriber Migration: In selecting the destination broker for a subscriber migration, different destination brokers will endure the same amount of additional outgoing load, but different destination brokers will have different additional

amount of incoming load. Therefore, the similarity-based migration technique selects the destination broker as the one that has the minimum increase of load when accepting the migrated subscriber. In other words, the migrated subscriber prefers to move to a broker that holds the largest subscription sharing with it. The degree of sharing between a subscriber and a broker is measured as the *subscription similarity score*, which is defined as the sum of data rates (λ's) of those shared subscriptions. However, in order to reduce the number of redundant migrations which involve the multiple migrations of a single subscriber back and forth among several brokers, the chosen destination broker should have the current load less than some threshold (maybe the average load of all brokers) in order to be able to accept a migrated subscriber.

4.4.3 Stage 3: Shuffle

The shuffle scheme is triggered when the system experiences an extreme skewed load distribution, e.g., when $cov > \gamma$ and $\mu > \beta$. Again, these γ and β values are determined by empirical studies. The extremely skewed load distribution indicates that the current mapping of subscribers to brokers is not appropriate. The shuffle process looks at the whole set of subscribers and their subscriptions to re-distribute them among available brokers to potentially produce an uniform load distribution across all brokers. We implement a simple greedy shuffle algorithm as presented below.

Greedy Shuffle Algorithm: All brokers start with no subscriber. Subscribers are then ranked by their individual load measured as the total amount of data rates for their subscriptions. We iteratively assign the heaviest subscriber to the current least loaded broker and the load of the broker is then updated by the equation 4.5. The process finishes when all subscribers are assigned to brokers.

4.5 Experimental Evaluation

We implement and evaluate our proposed multistage adaptive load balancing scheme against the small scale prototype BDPS system. We also create a large scale simulated BDPS system for thorough evaluation of our load balancing scheme under different real-world settings.

4.5.1 Prototype System Evaluation

The small-scale prototype system consists of one BDMS cluster, one BCS, five brokers.

Hardware Setup

The BDMS is implemented on a cluster of three Intel NUC nodes and the BCS and five brokers run on three other nodes. Each node has a four core i7-5557U CPU processor, 16 Gigabytes of RAM and 1TB hard drive. The nodes in BDMS cluster are connected via a Gigabit Ethernet switch where the broker network and BCS are communicated via Ethernet network. The four hundred simulated subscribers connect to the system from a desktop machine and via WiFi network.

An Emergency Application

Channel Name	Parameters	Channel Execution Period
Emergencies Of Type	Event Type	10 seconds
Emergencies At Location	Event Location	20 seconds
Emergencies Of Type at Location	Event Type, Location	30 seconds
Emergencies Of Type At Location With Shelter	Event Type, Location	30 seconds
Emergencies Near Me	User Location	10 seconds
Emergencies of Type Near Me	Event Type, User Location	10 seconds
Emergencies Of Type With Shelter Near Me	Event Type, User Location	10 seconds

Table 4.1: The list of channels

We leverage the hypothetical emergency notification application as the driving test case. The application defines seven channels with various channel execution periods. The channels take different sets of parameters, including parameters on *Event Type*, *Event Location*, and *User Location*. The channels are listed in table 4.1. The creation and definition of channels, data types, and datasets used in this and following chapters, are presented in detail in Chapter 3 and Appendix A. The application simulates eight emergency creators as data publishers who produce synthetic time series publications of emergency reports. These reports are fed into associated datasets in the BDMS cluster. The Opportunistic Network Environment simulator [50] is used to model the movements of end-users, the occurrence and travel paths of emergency events. Figure 4.3 demonstrates a snapshot of 400 subscriber locations and the locations and impact zones of four emergency events. Over an experimental run-time of

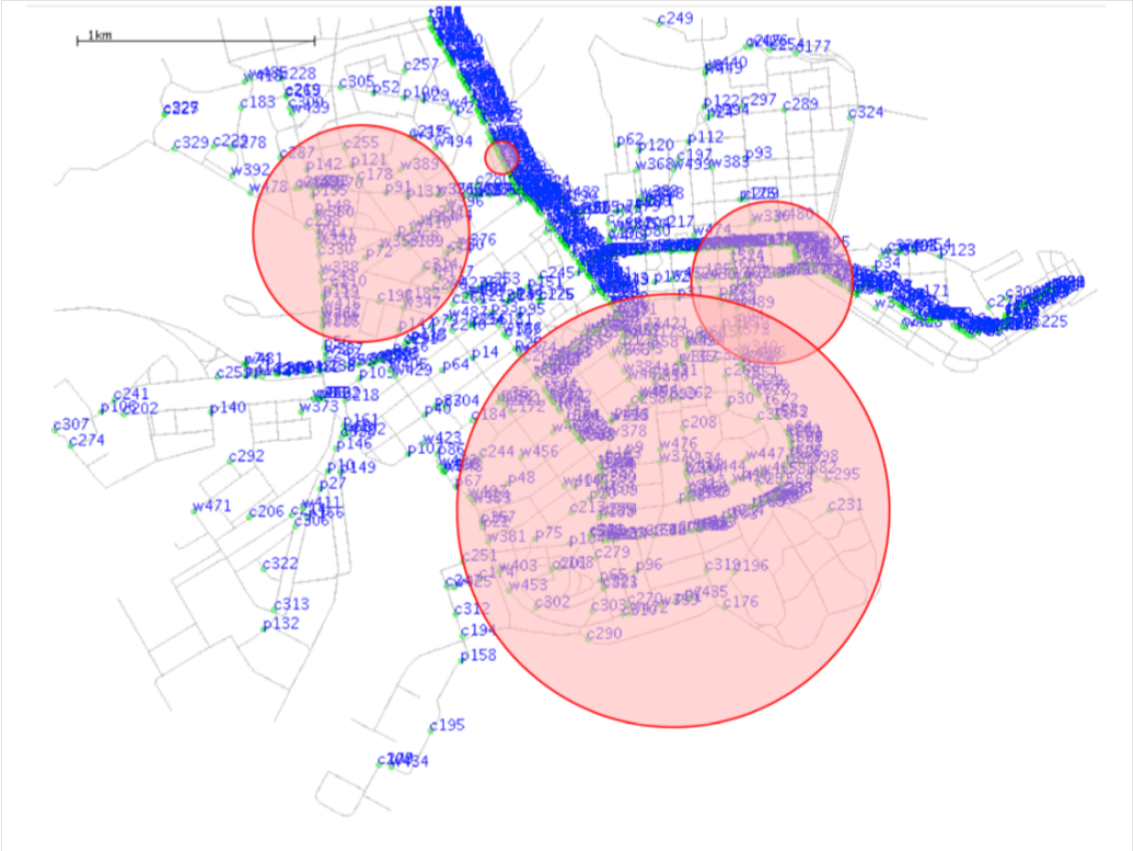


Figure 4.3: A snapshot showing locations of 400 subscribers and the occurrence of four emergency events

35 minutes, eight data publishers creates roughly 600 emergency reports of *landslide* events, 150 reports of *wind* events , 200 reports of *shooting* alerts, 200 reports of *flood* alerts, 2200 reports of *fire* events, 200 reports of *disease-outbreak* alerts, 600 reports of *earthquake* events, 6000 reports of *riot* events and the size of reports is in the range of (200, 700) bytes.

We store the time series actions of subscribers, such as logins, logouts, subscriptions, and un-subscriptions in a scenario file. This scenario file is then played back to model the interactions of subscribers with the system. Locations of subscribers are updated to the BDMS continuously when they move. We model subscriber logins to the system over the first four minutes, and model subscriptions over the next four minutes in an experimental run. Overall, the system produces around 600 back-end subscriptions and 2200 front-end subscriptions. We run the same experimental setup multiple times and apply different load

balancing schemes to compare the performance of these schemes. We then measure the data rates of subscriptions, and the loads of both subscribers and brokers, using a moving average, windowed over five minutes to avoid instantaneous peaks.

Prototype Experiment Evaluation

By applying our proposed load balancing techniques, we aim to minimize the maximum loads of brokers, as well as load imbalances, as represented by the coefficient of variation indicator. Therefore, we show effectiveness of our schemes on two performance metrics: *maximum broker load* and *coefficient of variation*. Cost is measured as the number of migrated subscribers needed. We show the effects of both dynamic migration and shuffle schemes, and demonstrate that our load balancing techniques work independently of initial placement policies. Table 4.2 includes abbreviations used in this chapter.

Abbreviation	Phrase
cov	Coefficient of Variation
No LB	No Load Balancing
NR	Nearest Broker Assignment
RR	Round-robin Broker Assignment
RND	Random Broker Assignment
LDM	Load-based Dynamic Subscriber Migration
SDM	Similarity-based Dynamic Subscriber Migration
GSH, SH	Greedy Shuffle

Table 4.2: Abbreviation

Evaluating effectiveness of the proposed multistage adaptive load balancing scheme: We study the efficiency of our load balancing scheme on three initial subscriber placement

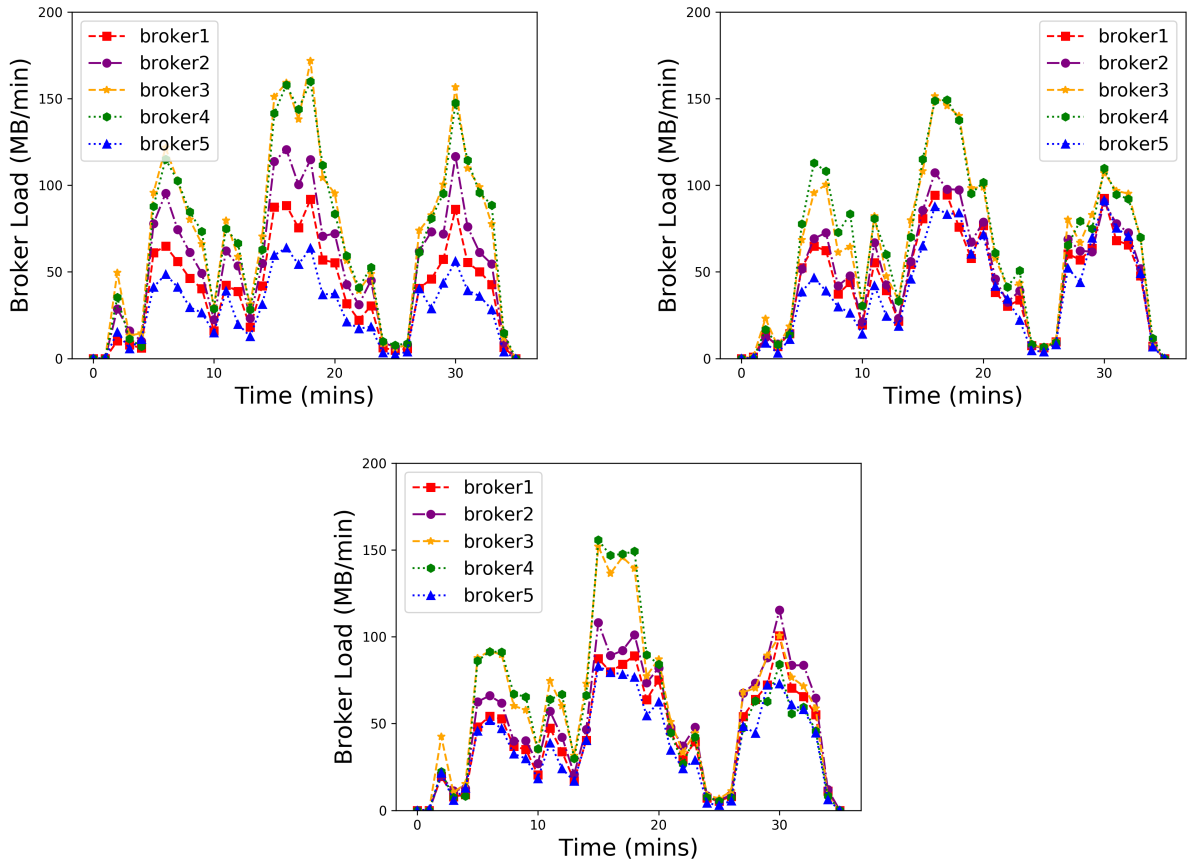


Figure 4.4: Broker load distribution (random broker assignment): (1) RND + No LB (2) RND + LDM (3) RND + SDM

policies including random broker assignment, round robin assignment, and nearest broker assignment.

Figure 4.4 presents the broker load distribution over an experiment run where the *random initial subscriber placement* is used, for various scenarios when: no load balancing scheme, load-based dynamic migration technique, and similarity-based dynamic migration technique is applied separately. Figure 4.5 presents measurements for the performance metrics including: maximum broker load, coefficient of variation, and total number of migrated subscribers. Since the load imbalance in this scenario is small, a few subscribers migrate and the shuffle is not invoked.

Similarly, Figure 4.6, 4.7 present the broker load distribution following the *nearest broker*

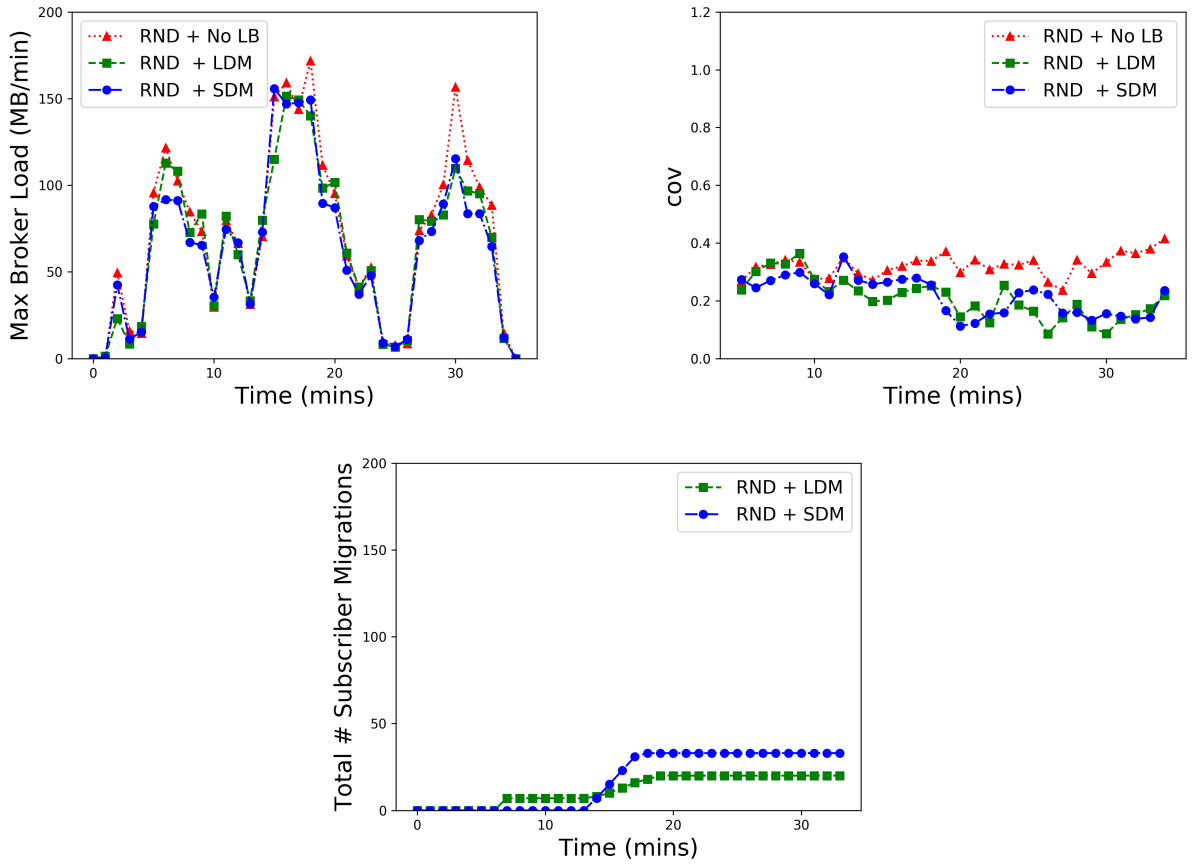


Figure 4.5: Performance metric measurements (random broker assignment): (1) max broker load (2) coefficient of variation (3) total # of migrated subscribers

placement policy when: no load balancing scheme, only a dynamic migration technique, and a combination of a dynamic migration and a greedy shuffle is applied separately. In this experimental setup, the nearest broker placement turns out to produce an extremely unbalanced load distribution because of the non-uniform distribution of subscribers geographically. This extremely skewed load distribution calls out the shuffle process. Figure 4.8 and Figure 4.9 respectively presents the performance metric measurements when a dynamic migration technique, and when a combination of a dynamic migration technique and a shuffle is applied. We conclude that our load balancing techniques are able to fix the extremely high load imbalances as with the nearest broker assignment, and reduce the slight load imbalances as with the random broker assignment. However, the required number of migrated subscribers is not the same.

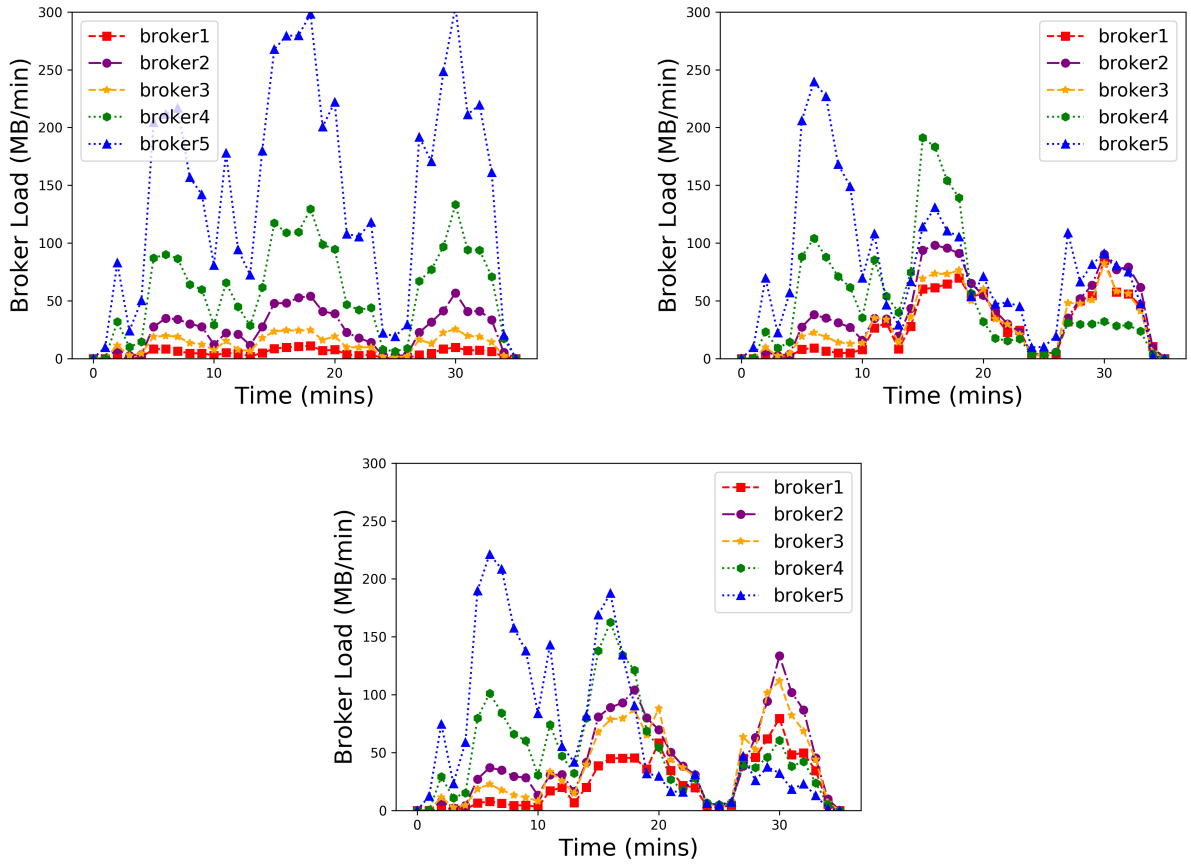


Figure 4.6: Broker load distribution (nearest broker assignment): (1) NR + No LB (2) NR + LDM (3) NR + SDM

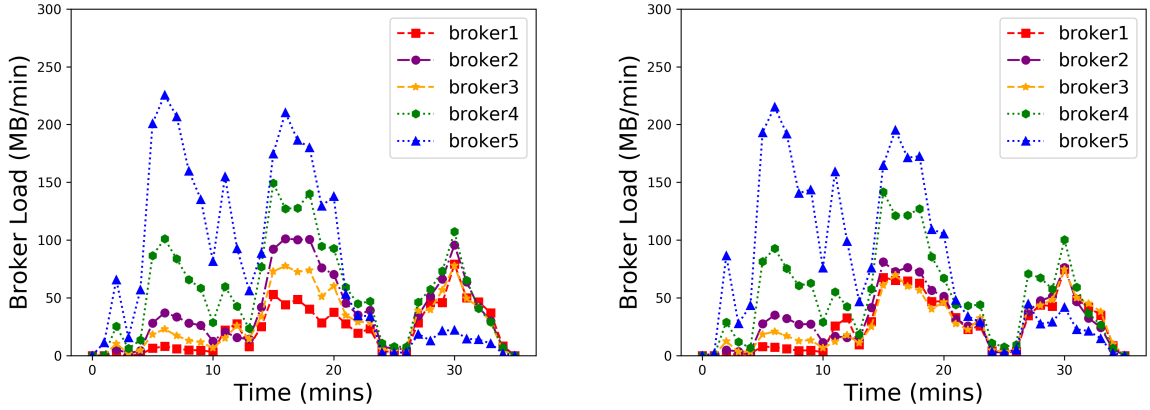


Figure 4.7: Broker load distribution (nearest broker assignment): (1) NR + LDM + GSH (2) NR + SDM + GSH

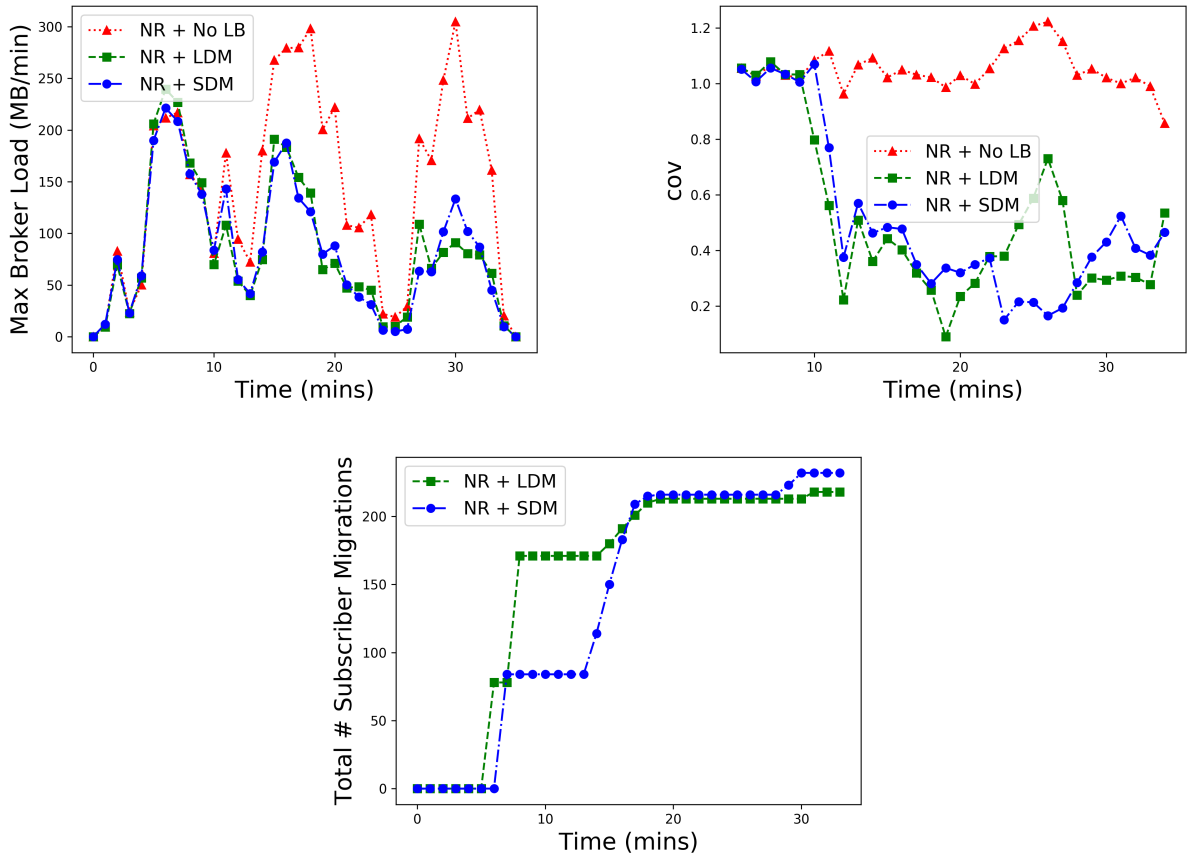


Figure 4.8: Performance metric measurements (nearest broker assignment and dynamic subscriber migration application only): (1) max broker load (2) coefficient of variation (3) total # migrated subscribers

Figure 4.10 compares the total number of migrated subscribers using load-based migration technique versus similarity-based migration technique among three subscriber placement policies. The more skewed load distribution, the more number of migrated subscribers is required. In this experimental setup, nearest broker assignment policy leads to highest broker load imbalance hence results in maximal number of required migrated subscribers.

Exploiting subscription similarity between migrated subscribers and destination brokers: Destination brokers are brokers which migrated subscribers migrate to. Similarity-based migration technique aims to leverage subscription similarity between migrated subscribers and the destination brokers to minimize the retrieval load. However, retrieval load

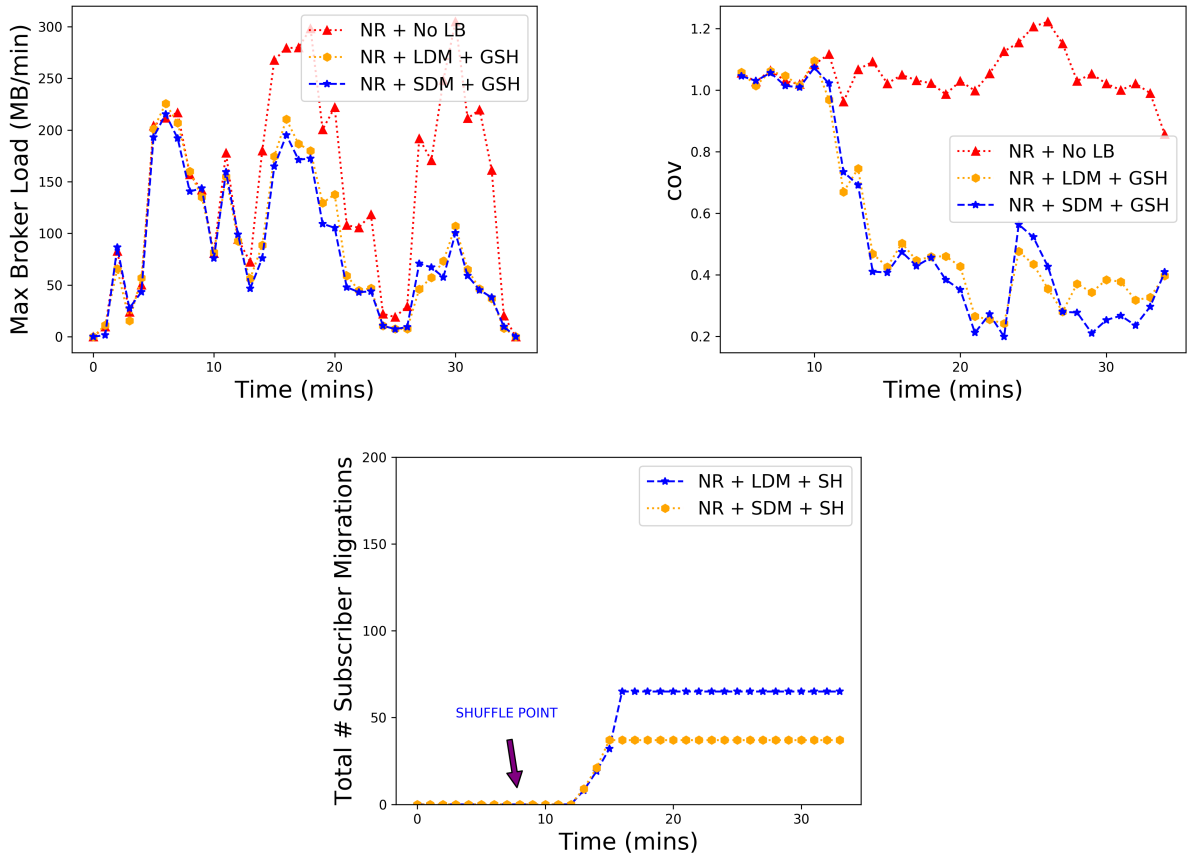


Figure 4.9: Performance metric measurements (nearest broker assignment, combination of dynamic migration and shuffle): (1) max broker load (2) coefficient of variation (3) total # subscriber migrations

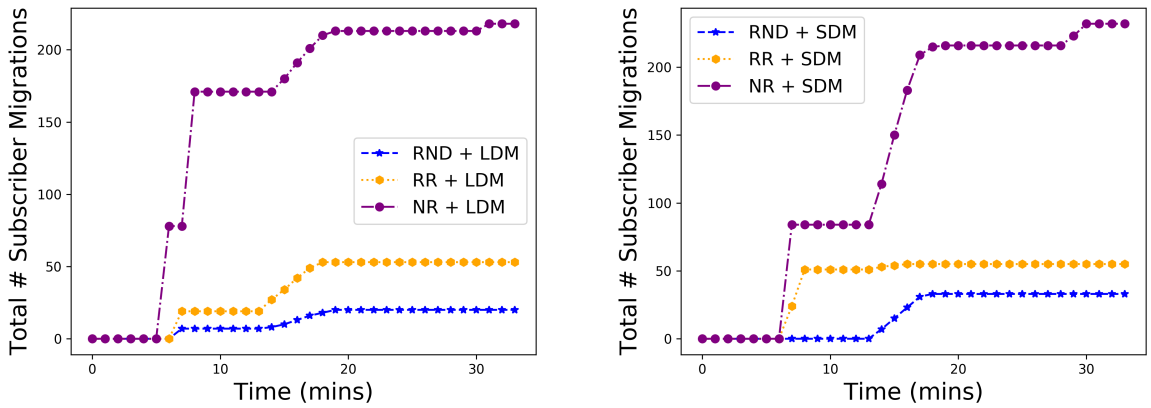


Figure 4.10: Total # migrated subscribers (RND, RR, NR placement policies): (1) LDM (2) SDM

is much smaller than delivery load at a broker. Hence, the performance of similarity-based migration and load-based migration techniques is similar as shown in Figure 4.5 and Figure 4.8.

Examining effectiveness of greedy shuffle technique: Fig. 4.9 (3) shows that a shuffle is triggered at an early stage when a load imbalance is detected, which helps fix the poor subscriber-to-broker assignment. The shuffle scheme reduces the required number of migrated subscribers in the future compared to the case when only a dynamic migration technique is applied.

4.5.2 Simulation Based Evaluation

To further evaluate our proposed techniques in a scaled up setting (beyond the scope of our prototype experiments), we use a simulation-based approach. We develop a simulator written in Python3 that mimics the messaging-level interactions among BDPS components: BDMS, brokers, BCS, and subscribers.

Simulation Setup: In our simulation setting, we model 10 brokers to support 10,000 subscribers. We create 10 channels having execution periods every 5, 10, 20, 30, 60 seconds and each channel has 100 distinct pairs of (parameter, value). Each back-end subscription has a notification data rate that follows a Normal distribution with a predefined mean and a standard deviation value. Subscribers creates a number, in the range from 10 to 30, of subscriptions, results in roughly 200,000 front-end subscriptions in total. Each simulation runs for half an hour where all subscribers login and create subscriptions in the first eight minutes. During the simulation run, we change the notification data rates of some back-end subscriptions to create a dynamic workload.

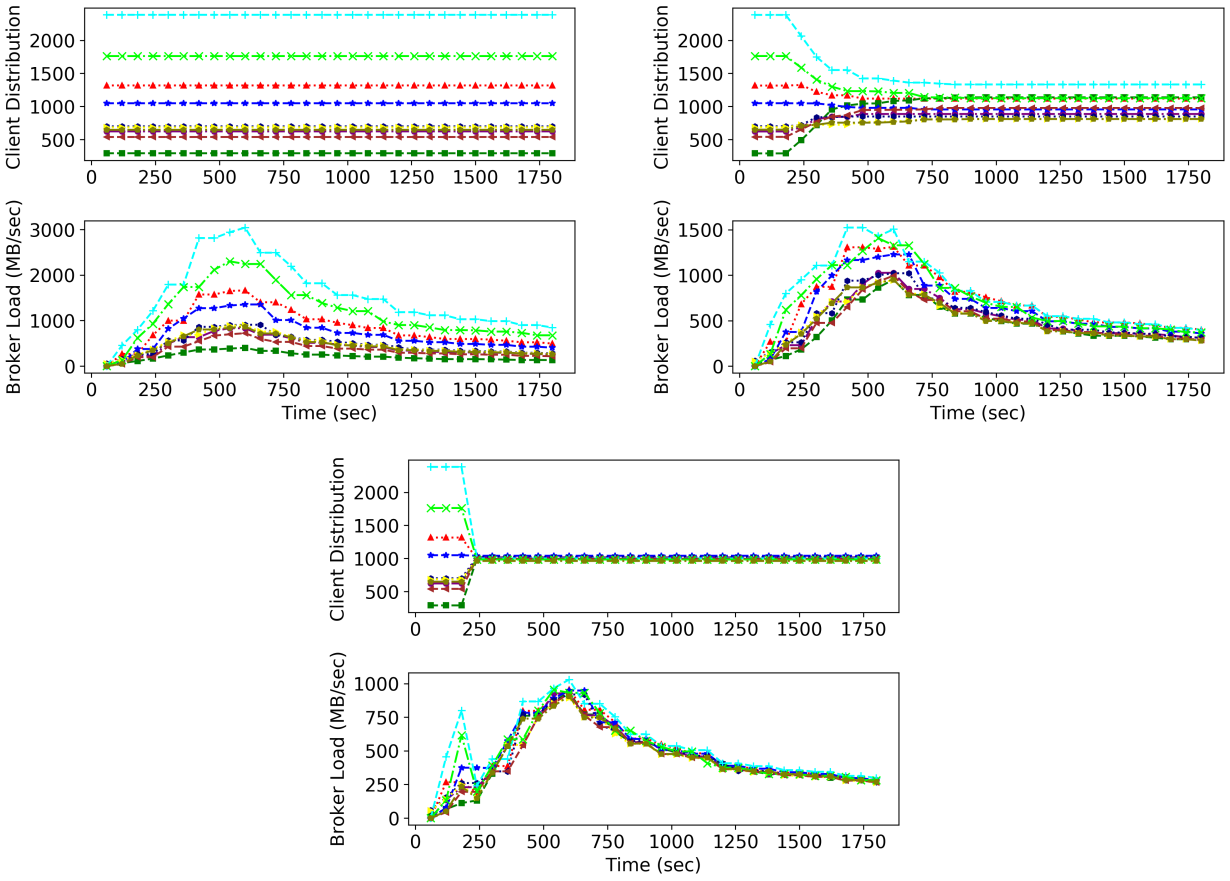


Figure 4.11: Broker load distribution: (1) NR placement + No LB (2) NR + LDM (3) NR + GSH

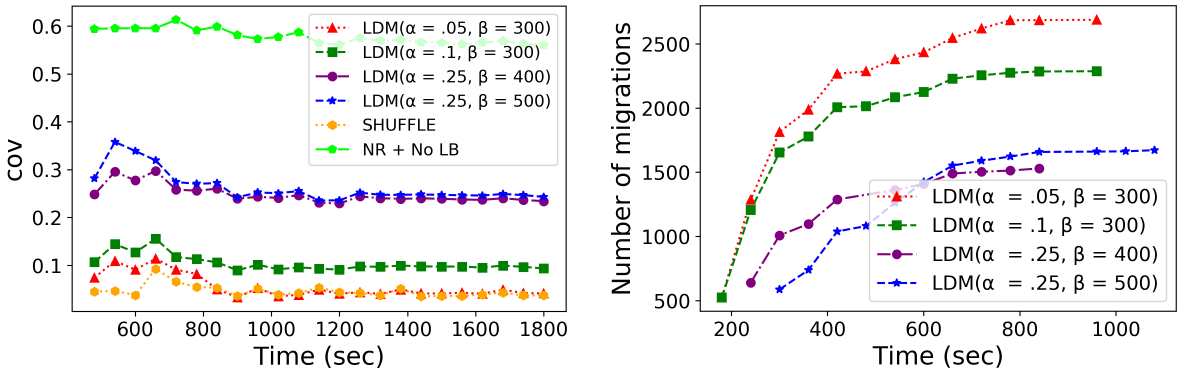


Figure 4.12: Evaluation of α and β values towards: (1) cov (2) number of migrations

Simulation Based Experimental Results We evaluate the load-based dynamic migration and shuffle techniques on an experiment with the nearest broker assignment policy. Figure 4.11 (1) presents the client distribution among brokers and the broker load distribution when BCS maps subscribers to brokers based on their geo-locations. The skewed distribution of subscribers geographically results in an unbalanced allocation of subscribers among brokers. There is a broker that is assigned more than 2000 subscribers while some brokers only need to support less than 500 subscribers. We can see highly varied loads among brokers in Figure 4.11 (1). We then apply the load-based dynamic subscriber migration, $\alpha = 0.15$ and $\beta = 300$ MB/sec, and the shuffle respectively, and show the results in Figure 4.11 (2) and Figure 4.11 (3). Both load-based dynamic subscriber migration and shuffle can fix the load imbalance eventually but the shuffle leads to a more balanced state and requires no further subscriber migration.

Sensitivity of threshold value selection: We study how the chosen threshold values, α and β, used in our proposed load balancing techniques, affects the load balancing performance metrics as shown in Figure 4.12. When we set the α and β values are very small, $\alpha = .05$ and $\beta = 300$ MB/sec, the load-based migration can eventually achieve a balanced broker load distribution as good as the shuffle does as shown in Figure 4.12 (1). The smaller the chosen values of α and β, the more frequent the system triggers the subscriber migration process.

This helps the system to achieve a more balanced loads but with a higher cost of number of migrated subscribers as shown in Fig. 4.12 (2). Therefore, the α and β values should be chosen carefully to avoid redundant subscriber migrations. The shuffle only comes into play when the system experiences an extremely skewed load distribution. The shuffle can quickly fix the subscriber-broker mapping to achieve an uniform load distribution as shown in Fig. 4.11 (3).

4.6 Conclusion

In this chapter, we propose, implement and evaluate a multistage adaptive load balancing framework. We demonstrate the effectiveness of our proposed load balancing schemes to address different levels of skewed load distributions. Our load balancing technique is composed of three phases: initial subscriber placement, dynamic subscriber migration, and shuffle. We account for subscription similarity in the way we calculate retrieval loads at brokers. We also exploit subscription similarity in our proposed similarity-base dynamic subscriber migration in which we try to migrate subscribers to brokers who have higher subscription similarities with the migrated subscribers.

Chapter 5

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

As we attempt to provide customized alerts at scale through the BDPS architecture, faulty components can have a serious impact on whether and when subscribers receive notifications. In this chapter, we explore fault tolerance techniques and address the challenges in supporting reliability and scalability in societal-scale BDPS systems. 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 a novel technique, called REAPS - a primary-backup fault tolerance framework that can handle different classes of broker failures including randomized failures and geographically correlated failures.

5.1 Motivation and Overview

Fault tolerance is a concern in any distributed system; especially 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) [46, 52, 28, 95]. We start by determining vulnerable points in the BDPS workflow to focus our efforts in designing resilience techniques. At one end of the workflow, the back-end big data platforms are typically designed to accommodate large workloads; we assume that they operate on a cluster of servers with built-in high availability techniques [43, 42, 44, 16]. The other end involves subscribers that may crash, disconnect, leave or intermittently interact with the system. Failures of individual subscribers are localized to the subscribers and their behaviors (not under BDPS control) are hence not likely to impact the overall reliability of the BDPS system. 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 backend BDMS systems. Hardware redundancy and active software replication methods are hard (and expensive) to realize when the systems must scale to a large number of geo-distributed users. Additionally, failure scenarios may differ; a natural disaster may induce geo-correlated failures of multiple brokers in a region while hardware failures within a broker may be random at best. Towards this end, in this chapter, we design and develop REAPS a low-overhead fault tolerant service for BDPS systems.

Key contributions of this chapter include:

- Design of REAPS, a primary-backup based fault tolerance service for BDPS that exploits the unique characteristics of the BDPS architecture. 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 sub-

scription similarity to reduce backup maintenance overheads.

- Design of a quasi-active state replication protocol that executes synchronous subscription state replication and asynchronous notification state replication for low communication overheads.

- REAPS validation via prototype implementation and measurement studies; extensive evaluation under a variety of failure modes via simulation.

Related Work Over the years, 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 [6, 15, 52, 28]. For example, link failures are handled by the introduction of redundant links in the physical network or via application layer multicast/broadcast techniques [52, 28]. State machine replication methods are used to address crash or byzantine failures[6]. Similarly, check-pointing and logging techniques are used for rapid recovery when failures occur. Publish subscribe systems may fail at different layers of the execution stack. At the application layer, publishers may fail to send notifications to the network, subscribers may fail to receive notifications from the network. The network layer may cause notification corruption, out-of-order delivery or unacceptable delivery latencies [49]. 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 [72, 17, 37]. Techniques to mend and adjust the overlay to cope with broker churn or failure [73, 49, 21, 75, 95], reconstruct routing states at the recovering broker or buffer messages at the parent of failed brokers for re-transmission [93] have been studied. 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 systems are not applicable in BDPS architectures. 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).

5.2 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 service. Traditional primary-backup replication techniques vary in the degree and type of replication [14, 15] including state machine or active replication, semi-active or leader-follower replication [11] and passive replication (Figure . 5.1)

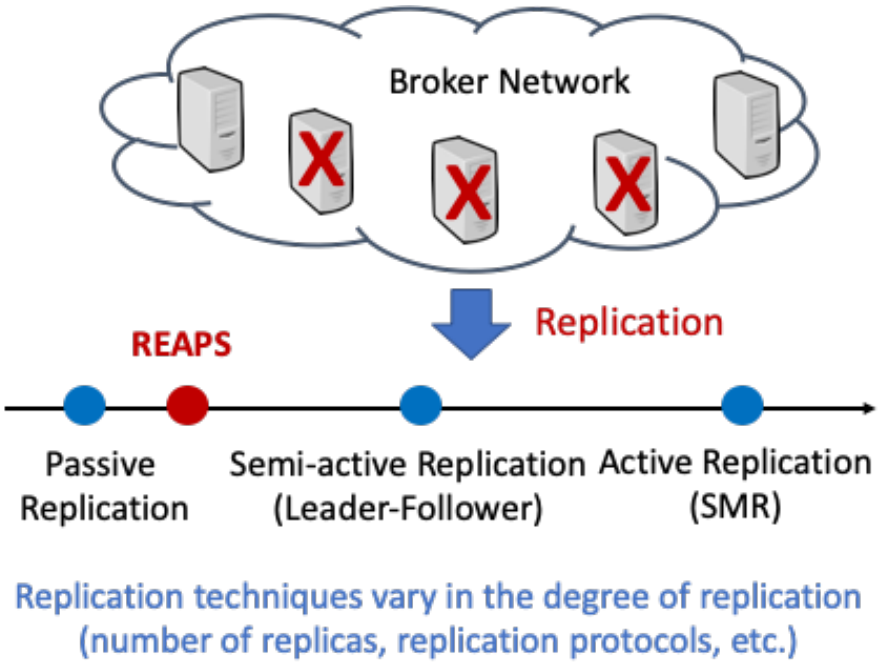


Figure 5.1: Replication Techniques

While the active approach may incur significant overhead when applied to a scaled-up BDPS

setting, the passive approach may cause significant recovery latency. We leverage the third variant that falls between purely passive and wholly active replication. i.e. *the semi-active or leader-follower approach* [11]. In contrast to traditional semi-active approaches (designed for computation-centric systems), BDPS systems are communication-centric. In REAPS, we propose a *quasi-active* fault tolerant technique to pro-actively maintain notification-related information, i.e partial state, for subscriptions at backup brokers.

The design goals for REAPS are as follows:

- Provide reliable notification delivery (i.e. eventual delivery)
- Minimize subscriber effort when assigned brokers fail, (e.g. subscribers must not need to recreate subscriptions)
- Support fast recovery of services under broker failure (minimize recovery latency or fail-over duration leading to minimum service disruption to end-users)
- Incur low additional overhead for fault tolerance (across layers of the pub/sub platform).

With REAPS, when brokers fail, the system detects failures and restarts service on replica brokers. For this, broker states must be instantiated/updated at the replica brokers to enable fast recovery of services. Under normal operation, primary brokers manage the BDMS and subscriber interactions; during this time, backup brokers remain synchronized through an adaptive state management protocol that is cognizant of the level of activity in the pub/sub framework. This enables quasi-active backups to concurrently occur and backup brokers to rapidly take-over functionality of the failed brokers when needed. We highlight three features in REAPS to ensure reliable notifications, reduce recovery latencies and diminish the overheads associated with fault-tolerance:

Allocate Backup Brokers: REAPS implements a two-level fault-tolerant framework to cope with two types of broker failures (section III) - (a) randomized failures where few brokers may fail randomly and independently 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. Each broker may serve as a backup for *multiple* other primary brokers. The ultimate goal of REAPS is to provide a fault tolerant service with no notification loss in case of failures (perhaps at some additional cost induced by duplicate notifications). In order to realize this, *primary states* to be replicated at the backups include information about subscribers, each subscriber's subscriptions and delivery of notifications. Note that actual notifications, which are stored at the big data back-end, are not replicated at the backup brokers. This reduces voluminous and unnecessary data transfers since past notifications often expire and are superseded by more recent ones.

Exploit Subscription Similarity: REAPS incorporates a subscription-aware approach in the selection and management of backups so as to minimize additional work done by backup brokers. Choosing a backup broker with overlapping subscriptions will incur reduced overheads for state maintenance and faster recovery. This is especially viable in the local backup scheme that aims to exploit subscription similarity among nearby brokers. The remote backup one aims to mitigate concurrent failures of brokers from independent geo-correlated failure zones.

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). In particular, the BDMS, primary and backup brokers cooperatively implement a proactive state-synchronization scheme to keep backups updated about subscriptions at the primary node as well as notifications being processed at the primary. Unlike active repli-

cation techniques, the backup remains passive and does not actively replicate the service, i.e. process or forward notifications to the subscribers to reduce overheads. The quasi-active approach in REAPS is adaptive and is able to exploit the rate at which publications arrive to tune state management overheads. It could also tune state synchronization overheads for each backup scheme individually. Local backups are proactive, they create and maintain subscriptions to the BDMS on behalf of the primary brokers ahead of failure time to shorten the fail-over process. A more aggressive state synchronization technique with higher overheads are implemented at the local backups to deal with randomized broker failures of higher probability occurring. Remote backups, in contrast are more passive and coordinate with the BDMS backend only when rare events of geo-correlated failures happen. In short, the local backup scheme in REAPS lies between the semi-active and passive replication. In the semi-active scheme, followers execute client requests completely, but do not propagate results to clients. However, local backup brokers only pre-create "backup subscriptions" but do not "execute" them (do not retrieve notifications from the BDMS, nor deliver notifications to end-subscribers). REAPS' remote backup scheme is close to passive replication.

Figure 5.2: REAPS Approach

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 subscribers' subscriptions for the fault tolerance service. The BCS uses this information for backup broker assignment decisions and coordinates the recovery process. Figure 5.3 shows our overall design for the fault tolerant service of BDPS systems in three phases. At the setup phase, subscribers register themselves with the BCS and get assigned their

own home brokers. They then connect and make subscriptions via their home brokers. During normal operation phase, brokers in the network send frequent updates of changes in their primary subscribers, primary subscriptions to the BCS. BCS performs the backup broker assignment based on the collected broker states and inform all brokers. Brokers then inform their attached subscribers. Each broker frequently replicates its primary state to its backups. The BDPS systems are dynamic in nature such as: subscribers may join or leave, subscriptions are created or withdrawn, notification rates may vary etc. The BCS may decide to re-assign backup brokers when the current backup assignment is not optimal as systems change or after failures. The backup re-assignment can be as simple as to re-run the backup assignment with current broker states or can be as dedicated as to develop techniques that consider the tradeoffs between the cost for performing re-assignment versus the benefit from the new optimal backup assignment. The BCS also monitors the broker network for failure detection and triggers the recovery phase when failures happen. Finally, during the recovery phase, the BCS informs backup brokers to take-over on behalf of failed primaries. Subscribers from the failed primaries re-connect to their corresponding backups. The BCS may need to run backup re-assignment 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 (Figure 5.2). We describe these problems and solution techniques in the next three sections.

5.3 Backup Broker Assignment in REAPS

In this section, we compute backup broker assignments to implement the REAPS approach. As indicated earlier, 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

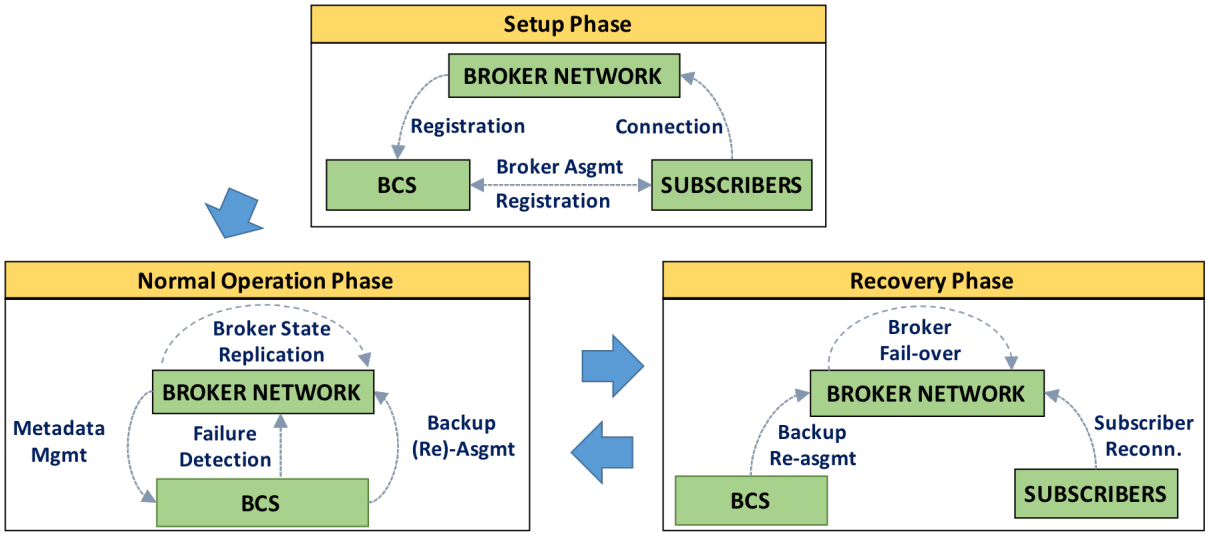


Figure 5.3: The Overall Fault Tolerance Approach

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. Locality plays a role in disaster events that may cause the failure of multiple nearby brokers in the affected regions. REAPS includes a comprehensive backup assignment strategy that considers the above issues. In REAPS, the broker network is divided into clusters called *availability zones* 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 (Figure 5.4). REAPS formulates backup assignment as an optimization problem aiming to minimize additional overheads cause by the fault tolerance/replication strategies.

REAPS Notation The following are notational conventions we use as we develop a mathematical formulation of the REAPS techniques. Consider a system with a total of M sub-

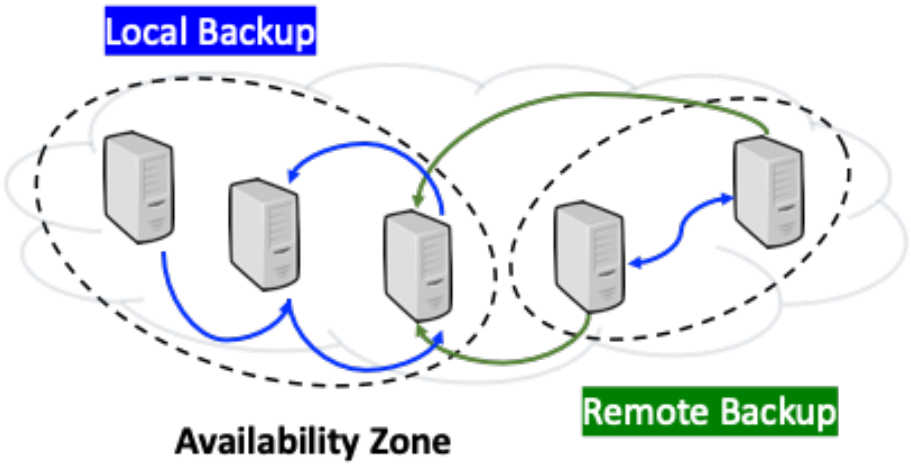


Figure 5.4: Replication Techniques

scribers, 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} be a binary indicator denoting whether broker i has subscription k ($z_{ik} = 1$) or not ($z_{ik} = 0$), c_{ik} denote the number of subscribers having subscription k at broker i. Let e_{ij} denote whether brokers i and j belong to the same cluster or not (e_{ij} equals 1 if they do, else 0). As per the broker network management, all the above information is maintained at the BCS that makes the broker backup assignment decisions. More precisely, 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. Once the assignment is made, the BCS passes the assignment information to the corresponding brokers.

5.3.1 Backup Broker Assignment Problem Formulation

Let x_{ij} be a binary decision variable that indicates whether broker i takes broker j as its local backup and y_{ij} be another set of binary decision variables for the remote backup assignment (y_{ij} is 1 if broker i takes broker j as its remote broker, else 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 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 subscribers and holds their subscriptions. Those subscriptions are called *primary subscriptions* and those subscribers are called *primary subscribers*. The primary subscriptions at a broker are said *active* and are replicated to the backups. A broker needs to continuously monitor new notifications for active subscriptions, retrieve and deliver them to corresponding subscribers. A broker being the backup for another broker would replicate primary subscriptions of that broker. Those replicated subscriptions are called *local backup subscriptions* at the local backup and *remote backup subscriptions* at the remote backup. As motivated by the measurement studies in prototype systems described in the experiment section, 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 monitor 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. These explain some part of the "quasi-active" in the overall REAPS framework.

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. That means, if two brokers happen to have higher subscription similarity (the higher number of identical subscriptions), assigning one as the backup for the other leads to considerably less overhead compared to other choices. The backup broker assignment optimizes this when assigning the backup pairs. A broker can be the local or remote backup for multiple other brokers. 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-overlap* local backup subscriptions across all brokers to minimize the so called *subscription overhead* during normal operation, whereas for the remote backup assignment, it tries to minimize the *largest* number of non-overlap remote backup subscriptions per broker to minimize the recovery latency. Moreover, in the process of backup assignment, brokers should not be over-committed, in which, their overall workload (regular plus backup) should be bounded within its capacity. Each broker specifies its limit, denoted as 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 for being chosen as a local backup for some other brokers is denoted as:

$$O_j = \sum_{k=1}^{N}(1 - z_{jk}) \times (1 - \prod_{i=1}^{L}(1 - x_{ij} \times z_{ik})) \tag{5.1}$$

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

$$R_j = \sum_{k=1}^{N}(1 - z_{jk}) \times (1 - \prod_{i=1}^{L}(1 - y_{ij} \times z_{ik})) \times \prod_{i=1}^{L}(1 - x_{ij} \times z_{ik}) \tag{5.2}$$

R_j takes into account the fact that broker j has already being assigned as the local backup for

some brokers and already pre-subscribed the local backup subscriptions. In the Equation 5.2, term $\prod_{i=1}^{L}(1 - y_{ij} \times z_{ik})$ denotes if any broker being remotely backup by j has subscription k or not; term $\prod_{i=1}^{L}(1 - x_{ij} \times z_{ik})$ denotes if any broker being locally backup by j has subscription k or not. The joint local and remote assignment problem is, therefore, to find assignment matrices X and Y so as to:

$$\min \sum_{j=1}^{L} O_j \textbf{ and } \min \max_{j=1}^{L} R_j \tag{5.3}$$

subject to:

$$x_{ij}, y_{ij} \in \{0, 1\}, x_{ii} = 0, y_{ii} = 0, \forall i, j \tag{5.4}$$

$$x_{ij} \leq e_{ij}, y_{ij} \leq 1 - e_{ij}, \forall i, j \tag{5.5}$$

$$\sum_{j=1}^{L} x_{ij} = 1, \sum_{j=1}^{L} y_{ij} = 1, \forall i \tag{5.6}$$

$$P_{j,X,Y} = \sum_{k=1}^{N} \lambda_k c_{jk} + \sum_{i=1}^{L}(x_{ij} + y_{ij}) \sum_{k=1}^{N} \lambda_k c_{ik} < D_j \tag{5.7}$$

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 5.5), (ii) each broker has exactly one local backup and exactly one remote backup (Equation 5.6), and (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 5.7) 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* [70, 67] and the remote backup assignment can be reduced to the *multi-processor scheduling problem* [51, 40] if these two sub-problems are considered independently. We attempt to solve two sub-problems independently, solving for local backup assignment followed by remote backup assignment: solving local backup assignment to minimize the overall subscription overhead; solving remote backup assignment later to take into account the pre-subscription of local backup subscriptions from the local backup assignment. Additionally, joint optimization problems are typically more complex (NP-hard) and repeated executions in dynamic situations is time-consuming. In the next section, we describe our proposed two heuristic algorithms for them.

5.3.2 Backup Broker Assignment Algorithms

We propose the following *Least Cost Selection* algorithm for the local backup assignment scheme and *Min Max Cost Selection* for the remote backup assignment scheme.

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-overlap 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).

Algorithm 2: 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**

$$P_{j,X_{(x_{ij}=1)}} = \sum_{i=1}^{L} x_{ij} \sum_{k=1}^{N} \lambda_k c_{ik} + \sum_{k=1}^{N} \lambda_k c_{jk} \tag{5.8}$$

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

$$\delta_j = \sum_{k=1}^{N} z_{ik}(1 - z_{jk}) \prod_{l=1}^{L} (1 - x_{lj} z_{lk}) \tag{5.9}$$

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

5.4 Broker State Management and State Replication in REAPS

In this section, we describe the formal broker state management methods. Note that each broker needs to maintain its own primary state about subscribers and their subscriptions, as well as the notification delivery. Additionally, a backup broker also needs to maintain the replicated states for its associated primary brokers. The process of state management varies across the local, remote backups and is also described here.

Algorithm 3: Min Max Cost Algorithm

```
1  Find: Y = {y_ij}
2  Initialize: Y = {y_ij = 0, ∀i, j = 1 ... L}
3  for i = 1, ..., L do
4      overhead = ∞ /* max number of non-overlap remote backup subscriptions per
           broker                                                              */
5      b = none /* remote backup broker selection                              */
6      for j = 1, ..., L and j ≠ i and e_ij = 0 do
           /* check potential workload for broker j if y_ij = 1               */
7          if P_{j,X,Y_{y_ij=1}} < D_j then
               /* check the max # non-overlap remote subscriptions per broker if
                   y_ij = 1                                                    */
8              if max_{l=1}^{L} R_{l,y_ij=1} < overhead then
9                  overhead = max_{l=1}^{L} R_{l,y_ij=1}
10                 b = j
11     y_ib = 1
12 return Y
```

5.4.1 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*. Each subscriber has a different behavior pattern: they may join the system or connect/disconnect from their home broker at a different time. Each subscriber subscribes to a different set of subscriptions. Therefore, we maintain the subscription state per subscriber and maintain the notification state per subscriber, per subscription. Each notification generated at the BDMS is annotated with a timestamp to denote its time of generation. 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 newest notification that

broker i has sent to subscriber m for the subscription k. When a previously disconnected subscriber re-connects to its home broker, the home broker must fetch and deliver all past notifications before sending any new notifications to the subscriber.

In our work, we refer to *state replication* as the process by which per-broker data-structures are replicated or synced with local/remote backups. Each broker i must maintain its own broker state as well as the backup broker states from its associated local/remote backup 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: $\Gamma_i, \Gamma_i^l,$ and Γ_i^r, across the brokers. We study when and how these states are copied to the required backup brokers to avoid notification loss and provide an upper bound to the possible notification duplication overhead during recovery. Here, the recovery refers to subscribers moving from their failed primary broker to their local or remote backup.

5.4.2 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, the subscriber state is immediately synced to the backups. When a subscriber creates a subscription or unsubscribes, the subscription state is also synced as soon as possible. Doing so will avoid the loss of subscription/subscriber information in case the primary broker fails. Since the subscriptions and subscribers rarely change, these synchronous updates do not incur much overhead on the broker. Upon receiving a subscription update, the local backup will create non-overlap or withdraw no longer exist backup subscriptions accordingly; the remote backup will update the backup subscription state, but will not make any changes to its current

subscriptions. 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. Since there is always a gap between the timestamp of new results and the timestamp of the last delivered notification, the broker will always try to catch up to the latest timestamp. In this context, 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. Thus, it is clear that $t_m^k \leq \tau^k$; 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. Since the notification delivery timestamps are not immediately synced with the backups, there is a gap between the currently recorded timestamp and the last synced timestamp. Let us denote 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 (Figure 5.6). We propose two techniques to replicate notification states: (a) periodic replication and (b) threshold-based replication. In the periodic technique, the notification state is replicated to the backup at a *sync* interval. On the other hand, in the threshold-based technique, the primary broker sends an update to its backups when the total volume of delivered but un-synced notifications to subscribers exceeds a pre-defined threshold, specifically: $\sum_{k,m}(t_m^k - \eta_m^k) \times \lambda_k > \beta$. We summary our broker state replication strategy in Figure 5.5.

We formalize the sub-routines for state management and state replication of the broker network in Algorithm 4. There are two RPC functions implemented on each broker, `l_sync` and `r_sync` (local sync and remote sync), that are called by primary brokers to sync their states to the local and remote backups, respectively. The routines show how states and their various components are updated.

Primary Broker State	Local Backup State Replication	Remote Backup State Replication
Subscriber Subscription State $S_i = \{m: \{k\}\}$	Synchronous Replication $S_i = \{m: \{k\}\}$	
Notification State $D_i = \{k: \{(m: t_m^k), \ldots\}, \ldots\}$	Threshold based vs Periodic Replication $D_i = \{k: \{(m: \eta_m^k), \ldots\}, \ldots\}$	
Back-end Subscription	Quasi-active Replication: replicated, but not "executed"	not replicated

Figure 5.5: Broker State Replication

5.5 Failure Model, Detection and Recovery in REAPS

The BCS is the central coordinator for our fault tolerance approach. It frequently receives metadata updates from brokers, and hence can recover its state soon after its failures. Once recovered, it can continue managing the broker network, facilitating new subscribers to join the system, and enable recovery for the broker network if needed. In REAPS, we assume that a BCS failures and broker failures are mutually exclusive events.

5.5.1 Failure Detection and Recovery

The BCS implements a failure detector with strong completeness - failed broker is eventually detected and eventual weak accuracy - there is a time after which some correct broker is never suspected [55]. 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. In our prototype implementation, the BCS uses the Tornado framework that supports 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

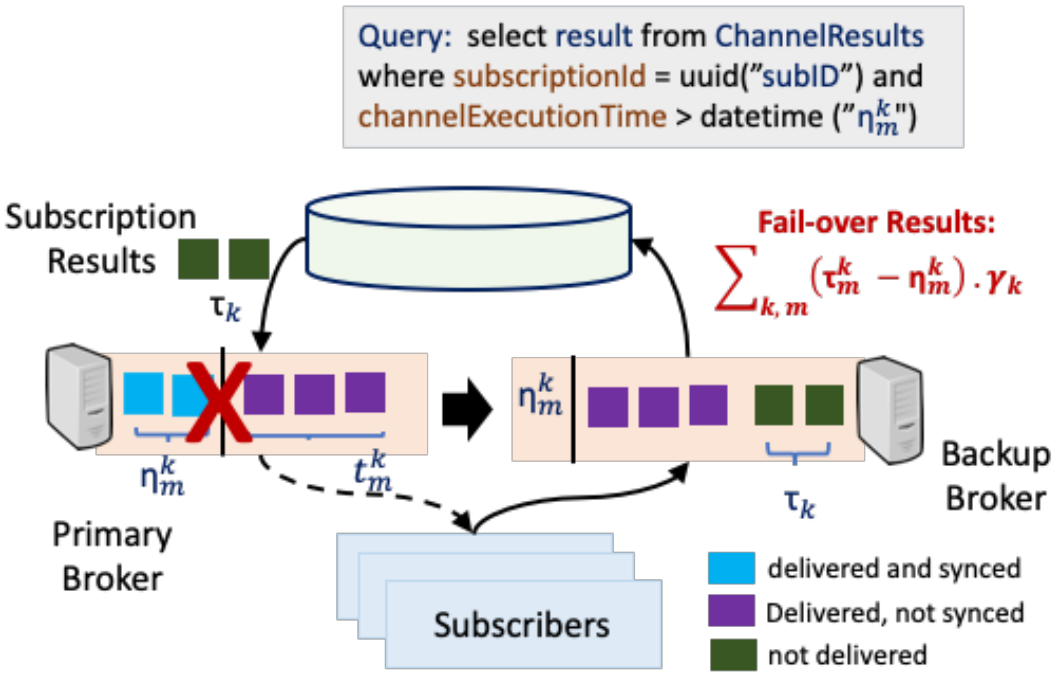


Figure 5.6: Notification State Replication and Retrieval of Fail-over Results at Recovery

a *time bound*. In this case, the BCS sends broker 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. We define a *local/remote fail-over* as the fail-over from a primary to a local/remote backup, respectively. For the local fail-over, we note that local backup subscriptions are al-

Algorithm 4: State Replication and Management at Broker i

```
1  sync-routine: /* called periodically or threshold-based                    */
2        lsync(i, Γ_i)
3        rsync(i, Γ_i)
4  on_notification(k, τ^k): /* on notifications from BDMS                    */
5        results = fetch (k, τ^k)
6        for online subscriber m subscribed to k:
7              push (results, m)
8              update D_i: t_m^k = τ^k
9  on_new_subscriber m:
10       lsync(i, Γ_i)
11       rsync(i, Γ_i)
12 on_new_subscription k from a subscriber m:
13       If k ∉ D_i: subscribe k /* send k to BDMS                           */
14       lsync(i, Γ_i)
15       rsync(i, Γ_i)
16 on_lsync (j, Γ_j): /* on a local update from broker j                     */
17       subscribe k1 ∈ S1
18       un-subscribe k2 ∈ S2
19       Γ_i^l[j] = Γ_j /* update broker state j at local backup           */
20       /* S1 = D_j \ {D_i ∪ Γ_i^l(sub)}                                    */
21       /* S2 = Γ_i^l[j][D_j] \ D_j \ ∪_{p#j} Γ_i^l[p][D_p] \ D_i           */
22       /* Γ_i^l(sub) = ∪_{j∈l(i)} D_j                                      */
23 on_rsync(j, Γ_j): /* on a remote update from broker j                     */
24       Γ_i^r[j] = Γ_j
```

ready pre-subscribed. Thus, the local backup just needs to retrieve the *fail-over notifications* and deliver them to the backup subscribers as shown in Figure 5.10. For the remote fail-over, the remote backup must first recreate remote backup subscriptions before it retrieves and delivers the fail-over notifications to the backup subscribers (Figure 5.7. The fail-over notifications are timestamped (η_m^k, τ^k]. To complete the fail-over process, the backup broker merges its backup subscribers with its own primary subscribers.

Notification Loss versus Duplication - A REAPS Tradeoff

In REAPS, we guarantee that subscribers do not lose notifications. Upon recovery from primary broker failures, the backups retrieve and deliver fail-over results to backup subscribers which cover the pending results supposed to be retrieved and delivered by the failed

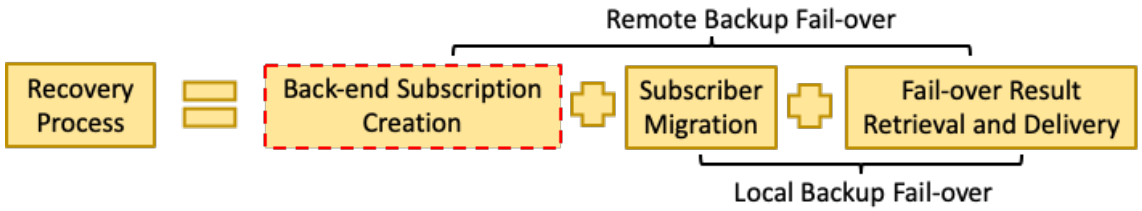


Figure 5.7: Recovery Process

primaries since $(\eta_m^k, \tau^k] \subseteq (t_m^k, \tau^k]$. However, each subscriber m receives a duplicate set of notifications in the window of $(\eta_m^k, t_m^k]$ for each of its subscriptions. Therefore, the number of duplicate results is bounded by $\sum_{k,m}(t_m^k - \eta_m^k) \times \lambda_k$.

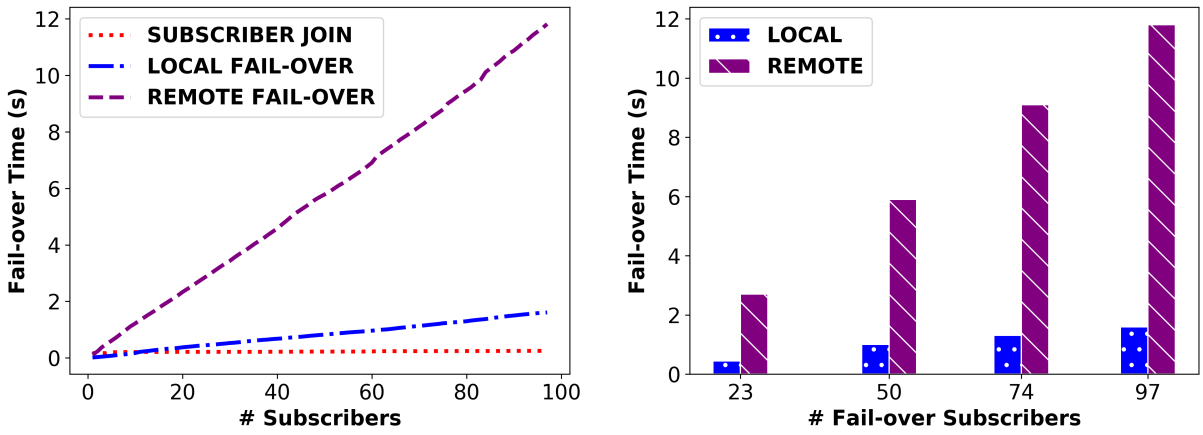
5.6 Experimental Evaluation

To evaluate REAPS on the prototype BDPS system and run simulation studies under different failure models.

5.6.1 REAPS Prototype Implementation and Measurement Study

Modeling Failures and Measurement Studies: We implement REAPS in a prototype BDPS system as a proof-of-concept and for real measurement studies. The broker network has four nodes partitioned in two clusters of two nodes. Each broker has the other broker in the same cluster as the local backup and one broker in the other cluster as the remote backup. We model 400 subscribers, each of whom is randomly assigned to brokers in the two clusters with one distinct subscription. Each subscription receives subscription results about emergency reports near the current location of each subscriber (every user subscribes to the *EmergenciesNearMe* channel which has the execution period of ten seconds). Thus, our system has a total of 400 distinct subscriptions, with each broker assigned to roughly 100 subscribers. To stress the BDMS, we model the extreme case where there were no

subscription similarities among brokers. After the fault-tolerance service is established, we examine the two local/remote fail-over process caused by a single broker failure. Fig 5.8a 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 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. Fig 5.8a shows that the local fail-over takes much less time than the remote fail-over; this is because the remote fail-over needs to create backup subscriptions. Fig 5.8b demonstrates that the fail-over duration is directly proportional to the number of migrated subscribers (23, 50, 74, 97) in both local and remote fail-over schemes. This conclusion motivates the design of REAPS.



(a) (b)

Figure 5.8: Prototype System: (a) Subscriber Migration Time versus Local Fail-over Time versus Remote Fail-over Time; (b) Varied # attached Subscribers at the Failed Broker

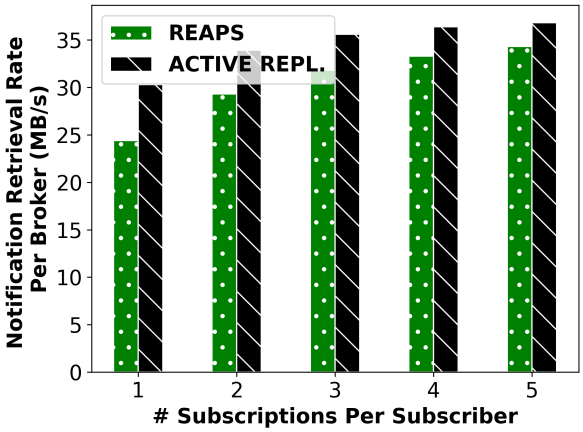
5.6.2 Simulation-Based Evaluation

We evaluate REAPS overhead and study performance in a simulation based approach under various broker failure models. The simulation models a BDPS system and mi-micks the messaging-level interactions among components. Our setup consists of 1 BDMS, 100

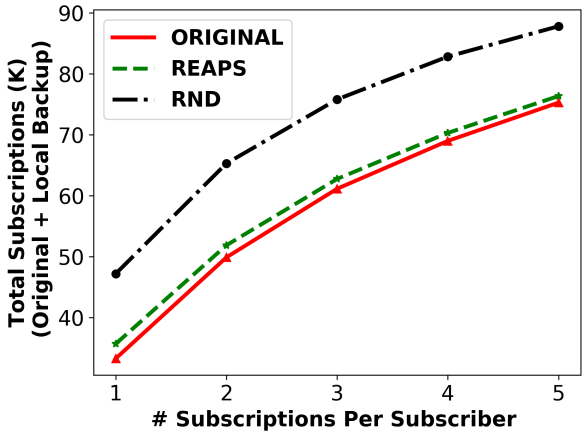
brokers, 100K subscribers, and 100 repetitive channels. Each channel has 10 distinct parameter value sets. Channel execution periods range from 10 to 600 seconds. The result size per subscription per channel execution time ranges from 10KB to 10MB for every channel execution. Finally, the broker network is partitioned into 3 clusters of 30, 30 and 40 brokers. Subscribers are assigned uniformly among the brokers. The subscription pattern from subscribers follows a Zipfian distribution where a large number of subscribers subscribe to a small number of popular subscriptions and a small number of subscribers subscribe to non-popular subscriptions. We consider the following types of fault tolerant overheads: i) *subscription overhead*; ii) *message overhead*; and iii) *notification overhead*.

REAPS vs. Active Replication: We evaluate REAPS against a naïve *active replication* approach. In the active replication, each subscriber connects and sends subscriptions to its home brokers as well as its backups. Backup brokers retrieve notifications from BDMS but do not deliver them to backup subscribers unless their primary brokers fail. At times of primary broker failures, the backup brokers in active replication approach do not need to send queries for retrieving fail-over results for backup subscribers since retrieving results for backup subscribers are already parts of the backup brokers' jobs during normal operations. Hence, the active replication technique incurs a constant notification overhead while REAPS incurs none. Fig 5.9 (a) shows the difference of notification retrieval rate per broker between REAPS versus active replication approach. However, since there is a subscription similarity among brokers, the notification overhead incurred by the active replication approach is not too high. When the number of subscriptions per subscriber increases, the subscription similarity among brokers increases, and the notification overhead in the active replication approach decreases.

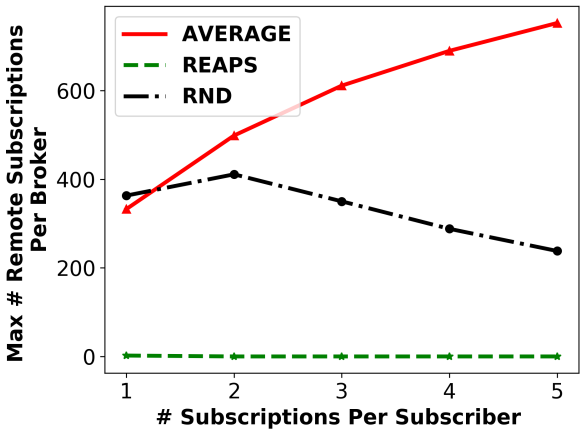
Evaluating Backup Broker Assignment Techniques: We compare the subscription overhead between REAPS and a random backup broker assignment against the total number of *original* back-end subscriptions. The red line in Fig 5.9 (b) represents the total number



(a) Notification retrieval overhead



(b) Local subscription overhead

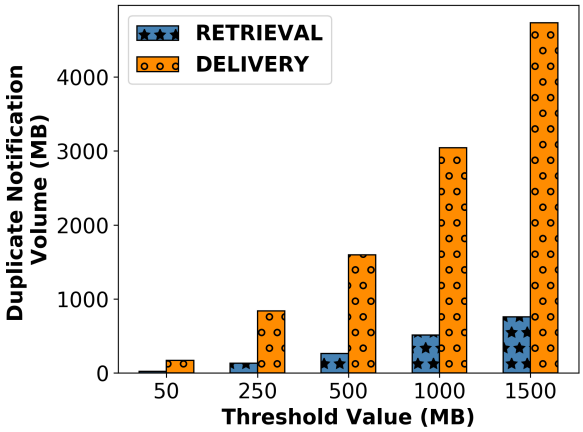


(c) Remote subscription overhead

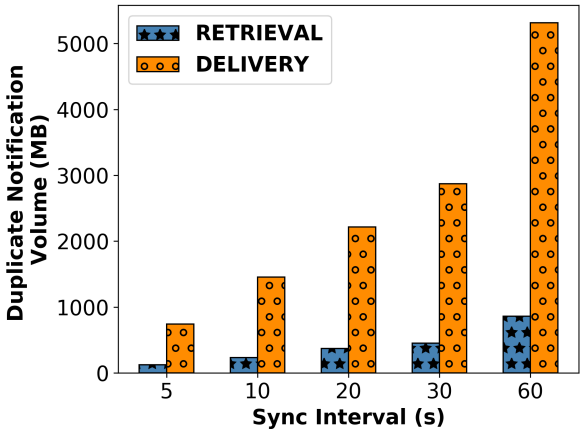
Figure 5.9: (a) REAPS vs. active replication approach
(b)(c) REAPS subscription overhead: local backup scheme & remote backup scheme

of original subscriptions across brokers. The green line represents the sum of total original subscriptions and total extra local backup subscriptions in REAPS scheme. The black line represents the sum of total original subscriptions and total extra local backup subscriptions in random backup assignment scheme. The gap between the green line and the red line represents the subscription overhead in REAPS scheme. Similarly, the gap between the black line and the red line represents the subscription overhead in random backup assignment scheme. Since REAPS assigns backup brokers which minimizes the total non-overlap local backup subscriptions; it produces much less subscription overhead. Figure 5.9 (c), on the other hand, shows the maximum number of non-overlapping remote backup subscriptions per broker in

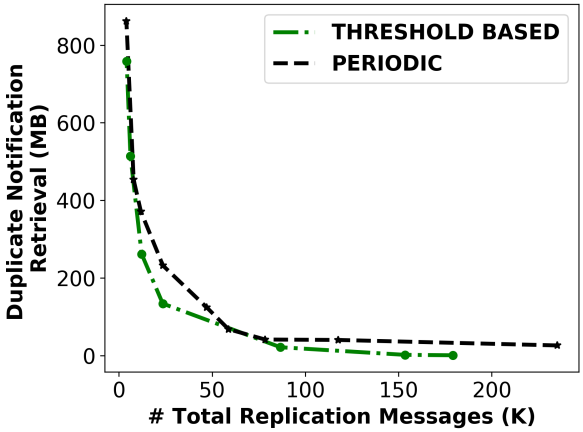
REAPS scheme (green line) versus a random remote backup assignment (black line). The red line represents the *average* number of original subscriptions per broker. REAPS minimizes the maximal number of non-overlapping remote backup subscriptions per broker; it leads to nearly zero non-overlapping remote backup subscriptions in this simulation setup. This fact can be explained that all remote backup subscriptions at a backup broker is covered by either the original back-end subscriptions at that broker or the local backup subscriptions which are pre-subscribed at that broker as it serves as the local backup for other brokers in the network. Should remote fail-overs are required in case of broker failures, the remote backup brokers do not need to create remote backup subscriptions for the failed primaries; which minimizes the fail-over times.



(a) Threshold based replication



(b) Periodic replication



(c) Threshold based vs periodic replication

Figure 5.10: REAPS: state replication evaluation for a single broker failure

Evaluating State Replication Methods: We evaluate the message overhead during normal operation and the volume of incurred duplicate notifications at failure recovery of the two proposed state replication techniques in a single broker failure scenario. Here, we define the *duplicate notification retrieval* as the volume of notifications that have been delivered to subscribers by a failed primary broker, but again be retrieved from BDMS and delivered to those subscribers by its local backup broker because the replicated notification delivery state at the backup broker is slower than the actual notification delivery state at the primary broker. On the other hand, the *duplicate notification delivery* is defined as the total volume of duplicate notifications received by all backup subscribers during the fail-over. Figure 5.10 (a) and 5.10 (b) demonstrate the volume of duplicate notification retrieval and delivery at different threshold values and periodic intervals. The higher the threshold value or periodic interval is, the less frequent the replication process is; which leads to the less message overhead but the higher volume of duplicate notifications retrieved by the backup broker and delivered to backup subscribers during the fail-over. Finally, Figure 5.10 (c) represents the relationship between the total number of replication messages during a simulation run versus the volume of retrieved duplicate notifications during the recovery of a single broker failure for the threshold-based and periodic replication techniques. Again, we can see as shown in this Figure, a higher message overhead yields a smaller duplicate notification volume during recovery. The threshold-based technique incurs a slightly smaller volume of duplicate notifications compared to the periodic replication technique at the same message overhead.

Multi-Failure Model: Finally, we evaluate REAPS against scenarios where multiple brokers fail. Figure 5.11 (a) 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 leads to a higher recovery latency. Similarly, Fig. 5.11 (c) demonstrates that a higher number of failed brokers across clusters results in a higher probability that a failed broker having no available backups.

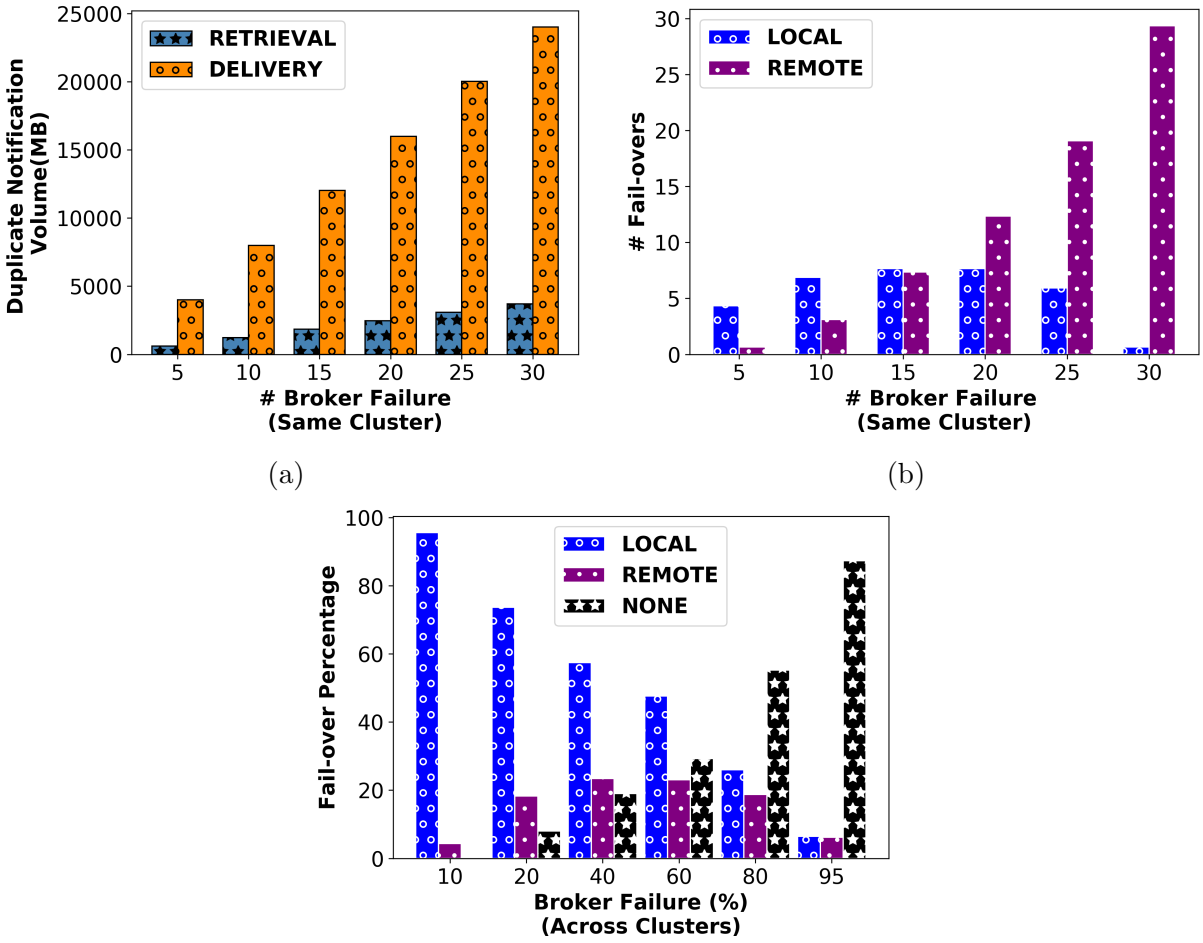


Figure 5.11: REAPS: multi-broker failure evaluation

5.7 Conclusion

In this chapter, we design and develop REAPS, a fault tolerance service for BDPS systems. REAPS exploits the hierarchical architecture of a BDPS system with a BDMS cluster and edge brokers as well as the characteristics of societal notification systems (geo and socially correlated interests) to address new tradeoffs between reliability, timeliness and scalability of notification systems.

Chapter 6

Notification Prioritization in Big Data Publish Subscribe Systems

Recent years have seen a dramatic increase in the number of subscription based applications and data-driven services that endusers subscribe to. Applications ranging from social media based interactions, e-commerce notifications and community services for public safety generate frequent notifications to devices to capture the attention of users. While such notifications can be incredibly beneficial to end-users, a constant stream of important and unimportant notifications can annoy users and cause them to withdraw/renege from notification services. The efficiency of a notification system depends on whether it can deliver the right message to the right person at the right time. Under normal workloads, a BDPS system can be designed to deliver all notifications to the subscribed end-users as they are generated with reasonable latency. In times of unexpectedly high workloads, the end-to-end platforms may fail to deliver critical messages to end-users in a timely manner due to limited resources. A key observation here is that some notifications are more important and valuable than others (e.g., emergency notifications versus social media notifications). Additionally, notifications may have deadlines and can be rendered useless if they get delivered late. For

example, timely notifications in emergency alert systems enable protective actions that can help reduce human injury, loss of life, and property damage.

6.1 Prioritizing Notifications

In this chapter, we develop techniques to prioritize notifications when brokers experience unexpectedly high workloads. Specifically, we show brokers can implement intelligent notification delivery scheduling so as to maximize the total benefits for all end-users of a BDPS system while guaranteeing fairness among them when there are unexpectedly high workloads at a broker, e.g., an increased volume of publications, an increased number of subscriptions, and/or an increased number of end-users. In this situation, the deployment of additional localized resources to accommodate increased workloads can be expensive and challenging. Techniques such as load balancing are not feasible when there are sporadic bursts of notifications targeted to end-users at a broker. Inherently, techniques to manage such sporadic notification bursts must be implemented within the broker so that timely and critical messages can be delivered to end users. In addition to the dynamic workloads generated on the channels, and dynamic system conditions (unpredictable network delays), users may have varying preferences for the arriving information and this in turn can alter the value of the notification to the end user. Our goal in this chapter is to design techniques to quantify utility/value of notifications to end-users and leverage this quantification to implement techniques to determine which notifications must be delivered (or dropped) and the order and time of delivery.

Chapter Road-map:

- We begin by developing a structured model for evaluating the value of notifications to end-users based on several factors, including channel characteristics, user preferences,

and delivery time. For this, we introduce two new concepts: channel value function and notification value function.

- We next model notification arrival and delivery processes at the broker level to design notification delivery scheduling policies to maximize user benefits and ensure fairness among end-users. For this purpose, we define metrics for evaluating user satisfaction and user fairness.

- We use the above models to develop notification prioritization algorithms. To evaluate the prioritization mechanisms proposed, we design a simulation-based approach to model the functioning of BDPS systems at scale , and validate the developed notification prioritization policies using simulation-based experiments with real-world use cases.

6.2 The Notification Prioritization Approach

Note that the value of a particular notification to a specific end-user depends on the nature and content of the notification, the time that it takes for the notification to reach the end-user, and the user's preference for that particular notification relative to other pieces of information (Figure 6.1). In this chapter, we aim to answer the following questions: What information is included in the notification? Is the notification still valid when delivered to the end-user? How much does the user like the notification content?

Figure 6.1: Notification Value to an End-user

To answer these questions, we employ prioritization notification as the key methods. Our notification prioritization approach operates in two steps(Figure 6.2). We first develop tech-

niques for quantifying notification values to end-users and then develop scheduling policies for notification delivery so as to maximize the total user benefits and ensure fairness among end-users.

As discussed earlier, The nature of a notification depends on the characteristics of the channel that generates it. The delivery times of notifications are affected by system workloads. User preferences vary according to user priorities, interests, and contexts. For example, parents prioritize emergency events at their children's schools. Commuters wish to know traffic information when they are on the road, but they may be more interested in news and entertainment events when they are at home. Investors have different preferences for different stock symbols. Citizens in New York are generally less interested in wildfire events in California. Below, we detail our approach to quantifying the value of notifications to particular end-users.

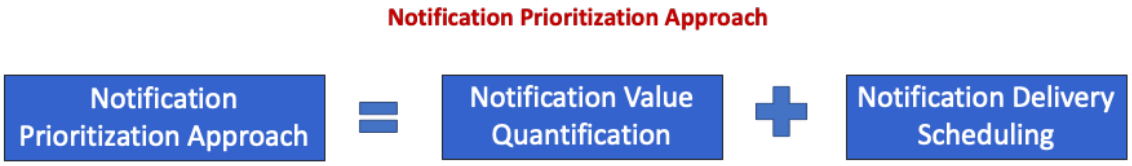


Figure 6.2: Notification Prioritization Approach

6.2.1 Quantifying Notification Value

Channel Value Function: From a system perspective (e.g., the design decisions of application administrators), the channels can be characterized by *channel importance*, which dictates the importance level of the notifications that they generate. In this study, we assume three levels of channel importance: *informational*, *important*, *critical*. For example, informational channels provide informational notifications such as notifications about sales events, advertisements, restaurant openings, and soccer match scores. Important channels provide information that can help end-users take actions and make better decisions in daily

life, such as notifications about stock trends, market analysis, business reviews for investors, traffic conditions for commuters, or school events for parents and students. Finally, critical channels are channels that provide notifications about critical events to end-users, e.g., emergency alerts during natural disasters.

In addition to channel importance, we introduce the concept of *channel type*. Channel type determines how the value of a notification from the channel decreases over time. We explore three types of channels, namely *deadline* channels, *update* channels and *decay* channels. A notification generated from a deadline channel has a constant value within the deadline, examples include sensor reading notifications. The notification deadline from a deadline channel is the channel execution period. However, a notification from an update channel is valid until the next notification is generated. Finally, the value of a notification from a decay channel decreases over time and turns to zero when it reaches the deadline. The channel type and the channel importance together determine the so-called *channel value function*.

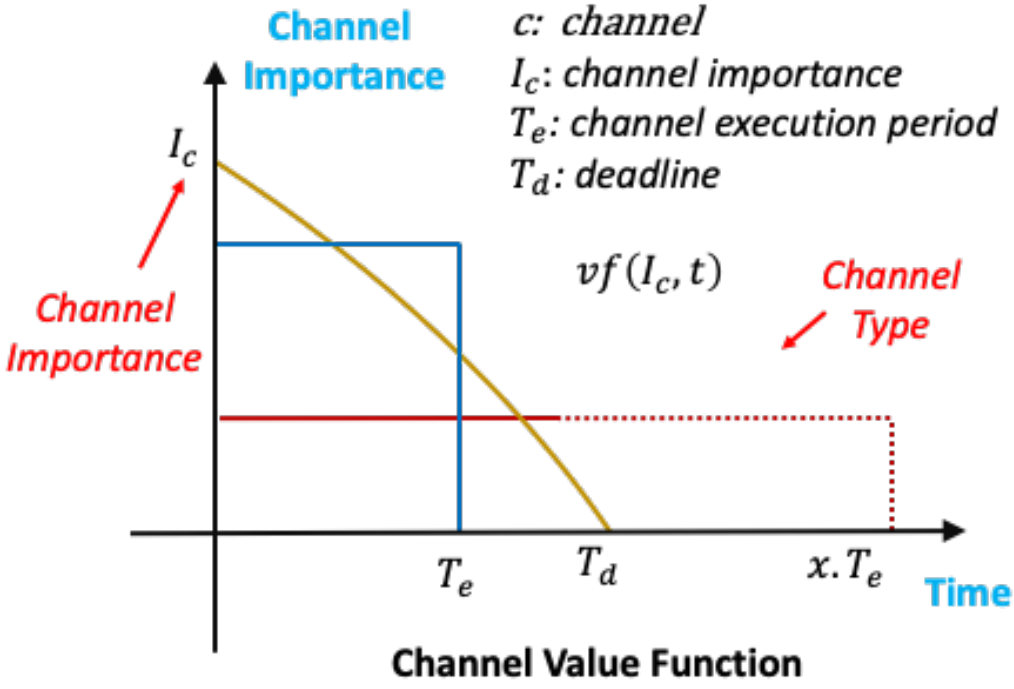


Figure 6.3: Channel Value Function

We define three channel value functions below, where c denotes the channel, T_d is the notification deadline, T_e indicates the channel execution period, and I_c represents the channel importance. The higher the importance level of a channel is, the higher its value I_c is. The channel value function is denoted as $vf(I_c, t)$. Channel value function determines the *base value* of its notifications at a particular time t, without considering user preferences. Since notifications from a channel are generated on a repetitive basis. The deadline of a notification starts at its generation time t_{gen}. Specifically, a notification that is generated at t_{gen} will expire at $t_{gen} + T_d$. We present below the abstract channel value function for each channel type without considering the generation time of specific notifications.

- Step Function - Deadline Channel

$$vf(c,t) = \begin{cases} I_c & t \leq T_d \\ 0 & t > T_d \end{cases}$$

- Open-ended Step Function - Update Channel

$$vf(c,t) = \begin{cases} I_c & t \leq T_d = x \times T_e \\ 0 & t > T_d \end{cases}$$

- Decay Function - Decay Channel

$$vf(c,t) = \begin{cases} \dfrac{I_c \times (1 - e^{\frac{t}{T_d}^{-1}})}{1 - e^{-1}} & t \leq T_d \\ 0 & t > T_d \end{cases}$$

Notification Value Function: We next quantify the value function of a notification n for subscription s from a channel c to an user u. The value of a particular notification depends on its channel value function, its delivery time to an end-user and the user preference. We

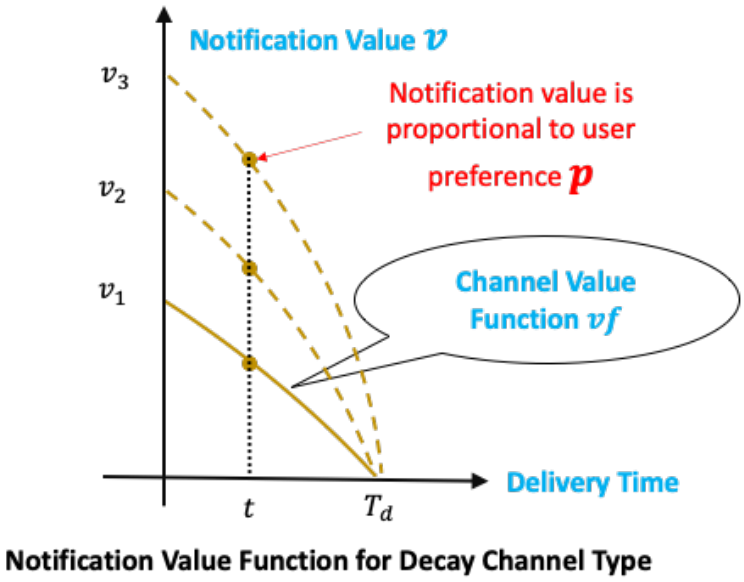


Figure 6.4: Notification Value Function

denote the preference of a particular user u to a specific subscription s from a channel c as $p(s, u)$. We define a *notification value function* to quantify the value of a notification over delivery time t to an end-user u, as follows:

$$V(n, s, u, t) = p(s, u) \times vf(I_c, t)$$

The equation shows that the higher the channel importance and user preference are, the earlier the notification is delivered to a subscriber, the higher the benefit the subscriber receives. Figure 6.4 represents the notification value function for a decay channel ($I_c = v_1$), where the value of a particular notification at a particular delivery time t to an user is proportional to the user preference.

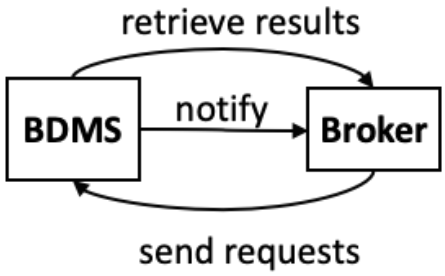


Figure 6.5: Broker Query Results from BDMS

6.2.2 Notification Arrival and Delivery Queuing Model

In the initial step described above, we designed a structured approach for quantifying notification value. We next develop a queuing-based model that uses the assigned notification values to schedule the delivery of notifications to end-users. Recall that a channel is executed repetitively at the BDMS to generate notifications for subscribers. When new results are generated, associated brokers are informed about the availability of new results by the BDMS (Figure 6.5). Each broker then retrieves available results for all users from the BDMS using a pull mechanism and inserts them into an *its arrival queue*. We refer to this as the notification arrival process at the broker. Notifications in a arrival queue are assigned values using the value quantification functions described above. Prioritization techniques are executed to order notifications and placed in the delivery queue at the broker to be delivered to end subscribers. Notifications that expire before being sent to all subscribers or notifications that have been sent to all subscribers are then deleted from the arrival queue. We refer to this step as the notification departure process. The above prioritization process is repeated as long as there are notifications to deliver. In this approach, notifications that have not been delivered to all subscribers but are still valid, are re-quantified for prioritization and considered for delivery in the next time unit.

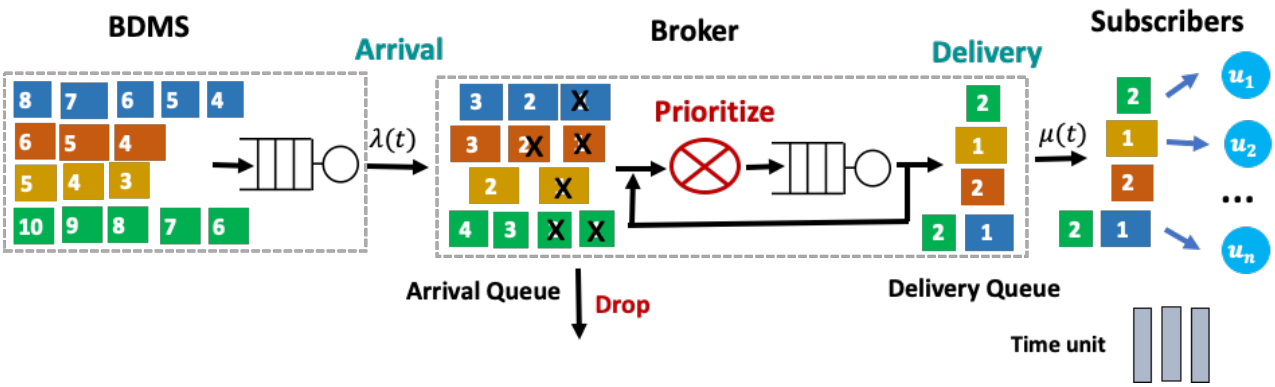


Figure 6.6: Notification Queuing Model

6.3 The Notification Prioritization Problem: Formulation and Algorithms

In this section, we develop notification prioritization techniques at the broker level to maximize the *total value* of notifications received by subscribers, The goal is to maximize the overall *user satisfaction* while ensuring a notion of *fairness* among subscribers. In this problem, we define *user satisfaction* as the ratio between the total received notification value versus the total expected notification value for a particular user. The prioritization technique determines which notification is delivered to which subscriber in which time unit.

6.3.1 Notification Prioritization - Problem Formulation

We define a notification delivery schedule $X = \{x_{n,s,u}(t), \forall n, s, u, t\}$, where $x_{n,s,u}(t) = 1$ when a notification n in the arrival queue for a subscription s is delivered to a user u at time unit t, and $x_{n,s,u}(t) = 0$ otherwise. We use $Q(t)$ to denote notifications in the arrival queue at time unit t. N represents all generated notifications and $N(t)$ denotes all notifications that have been generated until time unit t. We define a binary matrix $Y = \{y_{s,u}\}$ where $y_{s,u} = 1$ if user u subscribes for subscription s. We use $z_{n,s}$ to denote the size of notification n for

subscription s in bytes and $v_{n,s,u}(t)$ to indicate the value of notification n for subscription s to user u at delivery time t. Then we have,

$$v_{n,s,u}(t) = V(n, s, u, t)$$

We define $D_u(X, t)$ to denote the total notification value delivered to user u until time unit t as follows:

$$D_u(X, t) = \sum_{\tau}^{t} \sum_{(n,s) \in Q(\tau)} v_{n,s,u}(\tau) \times x_{n,s,u}(\tau)$$

$D_u(X)$ represents the total notification value delivered to user u across all time units, which is calculated as follows:

$$D_u(X) = \sum_{t} \sum_{(n,s) \in Q(t)} v_{n,s,u}(t) \times x_{n,s,u}(t)$$

The expected notification value to a user is defined as the value of the notification to the user at the time of generation. This is calculated with $v_{n,s,u}(t_{gen})$. Then, the total expected notification value received by a user u until time t can be calculated as follows:

$$E_u(t) = \sum_{(n,s) \in N(t)} v_{n,s,u}(t_{gen}) \times y_{s,u}$$

Finally, the total expected notification value received by a user u across all time units is defined as:

$$E_u = \sum_{(n,s)\in N} v_{n,s,u}(t_{gen}) \times y_{s,u}$$

Our notification prioritization problem aims to find a delivery schedule X that can ideally optimize multiple objectives - maximize total notification value delivered to each subscriber, maximize the total user satisfaction over all subscribers while ensuring fairness across subscribers. The notion of fairness among subscribers can be represented in multiple ways. We considered the following three possibilities - (1) maximizing the minimum user satisfaction; (2) minimizing the gap between maximum user satisfaction and minimum user satisfaction; or (3) minimizing the variance of user satisfaction. We assume that the system has m subscribers. We cast our problem to an optimization problem that aims to maximize the sum of average subscriber satisfaction calculated as $\frac{1}{m} \times \sum_u \frac{D_u(X)}{E_u}$, and minimum subscriber satisfaction calculated as $\min_u \frac{D_u(X)}{E_u}$.

We formalize the optimization problem as follows:

Find schedule $X = \{x_{n,s,u}(t), \forall n, s, u, t\}$

$$\max_X \left(\frac{1}{m} \times \sum_u \frac{D_u(X)}{E_u} + \min_u \frac{D_u(X)}{E_u} \right) \tag{6.1}$$

subject to:

$$\sum_u \sum_{(n,s)\in Q(t)} x_{n,s,u}(t) \times z_{n,s} \leq \beta, \forall t \tag{6.2}$$

$$\sum_t x_{n,s,u}(t) \leq y_{s,u}, \forall (n,s) \in N, \forall u \tag{6.3}$$

The constraints include (i) the total size of notifications scheduled for delivery in one time

unit is less than the broker capacity - outbound bandwidth (Equation 6.2); (ii) each notification n for a subscription s is delivered to a subscriber at most once if the user u subscribes to subscription s (Equation 6.3).

The above prioritization problem is an NP-hard. Here, the Knapsack problem can be reduced to a special case of the prioritization problem with one user and one time unit duration. We hence design and explore a set of heuristic algorithms for notification prioritization and delivery with different objectives, including enhancing notification value, user satisfaction, fairness and combinations of these criteria. We present and evaluate multiple such possibilities in detail below.

6.3.2 Notification Delivery Scheduling Algorithms

We next present a family of prioritization algorithms - where each algorithm emphasizes a different prioritization objective that drives notification delivery. In each prioritization technique, each broker selects notifications from the arrival queue for possible delivery in every time unit. The selection process for each time unit at a broker terminates when all notifications in the arrival queue are selected for delivery to all subscribed users, or when the broker reaches its bandwidth capacity (the total size of all selected notifications reaches the broker outbound bandwidth constraint).

- **High Value - HiVal**; This algorithm prioritizes notifications with highest values to end-users. In this prioritization technique, notifications in the arrival queue are sorted and chosen in the descending order of value to end-users until the broker bandwidth capacity (Equation 6.4) has been reached.

$$\arg \max_{(n,s)\in Q(t),u} v_{n,s,u}(t) \tag{6.4}$$

- **High Satisfaction - HiSat**; This algorithm prioritizes notifications to maximize total user satisfaction. The HiSat algorithm prioritizes notifications in the descending order of user satisfaction (actual value/expected value) as defined earlier.

$$\arg \max_{(n,s)\in Q(t),u} \frac{v_{n,s,u}(t)}{E_u(t)} \tag{6.5}$$

- **High Balance - HiBal**: This algorithm prioritizes notifications that achieves a balance between the maximum total value of delivered notifications and maximum user satisfaction (Equation 6.6). Let η be a factor to vary the prioritization weight of the total value of delivered notifications. In our experiment, we set η equal to the number of subscribers attached to the broker. Intuitively, this puts equal weight on both total value of delivered notifications and user satisfaction.

$$\arg \max_{(n,s)\in Q(t),u} \eta \times \frac{v_{n,s,u}(t)}{\sum_u E_u(t)} + \frac{v_{n,s,u}(t)}{E_u(t)} \tag{6.6}$$

- **High Loss - HiLoss**: This algorithm prioritizes notifications that would incur highest value loss if delayed to the next time unit.

$$\arg \max_{(n,s)\in Q(t),u} \Big(v_{n,s,u}(t) - v_{n,s,u}(t+1) \Big)$$

- **High Fair - HiFair**; This algorithm prioritizes notifications to maximize the minimum user satisfaction. It selects notifications so as to increase minimum user satisfaction.

$$\arg \max_{(n,s) \in Q(t)} v_{n,s,u}(t) \arg \min_{u} \frac{D_u(X,t)}{E_u(t)} \tag{6.7}$$

- **Random - Rand** algorithm selects notifications randomly from the broker arrival queue to deliver to attached subscribers and is used primarily for comparison during evaluation.

6.4 Simulation-based Evaluation

We use a simulation-based approach to evaluate our proposed prioritization algorithms. Figure 6.7 shows the model of our simulation which is written in Python3.

6.4.1 Simulation Setup

The simulation models an actual BDPS system which consists of a BDMS data cluster, a BCS, a network of (ten) brokers. In our simulated setting, we model a range of real world applications that generate notifications with varying levels of importance and urgency. Specifically, We create thirty channels equally distributed across the 3 importance levels. Accordingly we have ten informational channels, ten important channels, and ten critical channels. The channels have a uniform distribution over three channel types described earlier including - deadline channels, update channels and decay channels. Each channel has fifty different pairs of (parameter, value) combinations to represent a wide range of notification possibilities. We hypothesize a community of thirty thousand subscribers.

These subscribers are categorized into three groups that represent users with varying interests and roles. Our simulated community includes ten thousand young adults, fifteen thousand adults and five thousand emergency responders. Each subscriber creates ten front-end subscriptions. The young adult population subscribes to channels following the Zipfian distribution on [*informational*, *important*, *critical*]. This implies that young-adults are more likely to subscribe to information channels as compared to important and critical channels. Specifically, 74% front-end subscriptions from young adults are to informational channels, 18% front-end subscriptions to important channels and 8% to critical channels. Similarly, adults subscribe to channels following the Zipfian distribution on [*important* (74%), *critical* (18%), *informational* (8%)]. The emergency responders subscribe to channels following the Zipfian distribution on [*critical* (74%), *important* (18%), *informational* (8%)].

6.4.2 Simulation Model

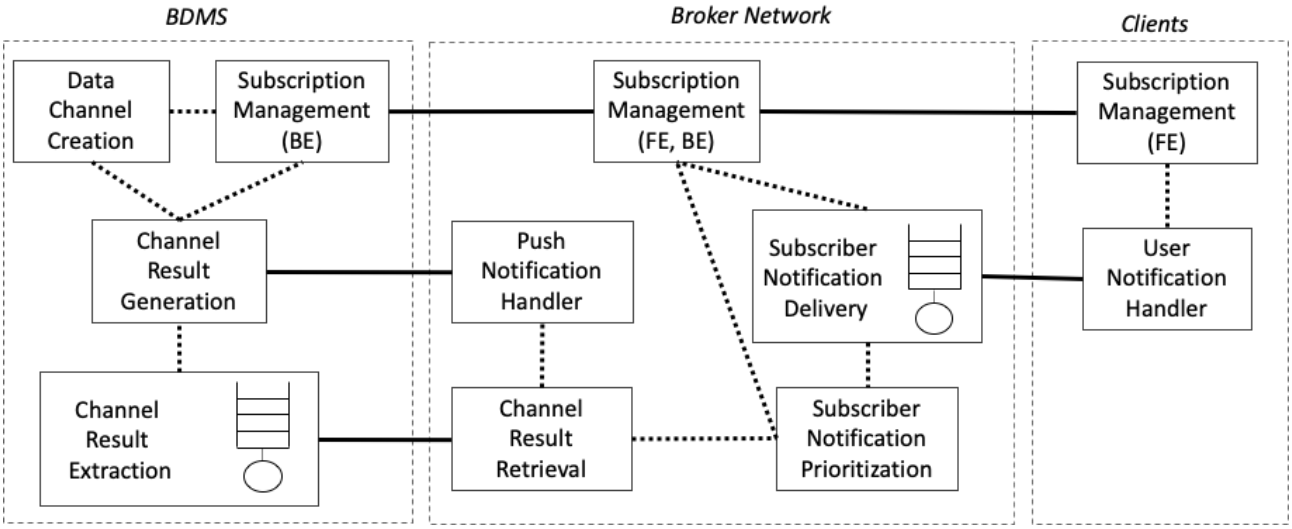


Figure 6.7: Simulation Model

We assume that the client side software at the user-end has a *Subscription* module which sends front-end subscriptions to the attached broker. Each broker has a *Subscription Management* module which receives front-end subscriptions from end-users and creates distinct

back-end subscriptions to the BDMS. The BDMS has a *Subscription Management* module which manages back-end subscriptions from brokers. The channels are created in BDMS in the module *Data Channel Creation*. Channels are executed periodically against the back-end subscriptions managed in the *Subscription Management* module to generate channel results for brokers in the module *Channel Result Generation*. When new channel results are available, associated brokers are notified by push notifications. The broker has a *Push Notification Handler* to handle the push notifications from BDMS and issues appropriate queries to retrieve channel results from BDMS in the *Channel Result Retrieval* module. The results retrieved from BDMS are prioritized in the *Subscriber Notification Prioritization* module (based on user preferences from *Subscription Management* module) and inserted into delivery queues for dissemination to subscribers in the *Subscriber Notification Delivery* module. In this simulation, we model the latency in obtaining/retrieving channel results from BDMS to brokers since multiple brokers may simultaneously send queries asking for results from BDMS. We describe below our measurement study using the prototype BAD BDPS system - we specifically conduct a set of latency measurements in retrieving channel results from BDMS to brokers. Note that latency may also be incurred in disseminating notifications from brokers to subscribers due to constraints on the broker outbound bandwidth - here our prioritization algorithms play the deciding role in determining which notifications are prioritized and delivered before others for maximum user satisfactions and benefits.

6.4.3 A Measurement Study using the BAD Platform

In order to approximately model the latency in retrieving channel results from BDMS to brokers, we conduct a prototype measurement study. We install a BDMS - AsterixDB version 0.9.5 on a MacOS with Apple M1 Pro CPU, 16 GB RAM and 500 GB Flash Storage. We run a simulated Tornado broker server running on the same machine. The broker sends a set of queries asking for results from BDMS. We measure the round trip time since the

broker starts to send the first query until the broker receives the returned results for the last query. We measure two scenarios where the result size for each query is of 88 bytes and 9 Kb respectively (Table 6.1 and 6.2). In our next measurement study, the broker sends only one query for retrieving channel results from BDMS. We measure the round trip time of the query with varied returned result size since the broker sends the query until the broker receives the results (Table 6.3). As we can observe from Table 6.1 and 6.2, the large the number of concurrent queries sent to BDMS, the longer it takes for BDMS to return all requested results to brokers. As shown in Table 6.3, the larger the returned result size, the longer it takes for the broker to receive the results from BDMS. Three graphs in Figure 6.8 are drawn for the measurements from Table 6.1, 6.2, and 6.3 separately. In the simulation, for simplicity, we assume the same result size of 88 bytes for all subscriptions from all channels. We use the measurement from Table 6.1 to model the latency for a broker in obtaining results from the BDMS based on the number of concurrent queries that are retrieving results at a particular time.

# Channel Result Queries	Time (ms)
10	40
100	350
300	1000
500	1600
800	2700
1000	3200
1500	4500

Table 6.1: Result size of 88 bytes

# Channel Result Queries	Time (ms)
10	200
100	1600
300	4800
500	8800
800	12800
1000	16600

Table 6.2: Result size of 9 Kb

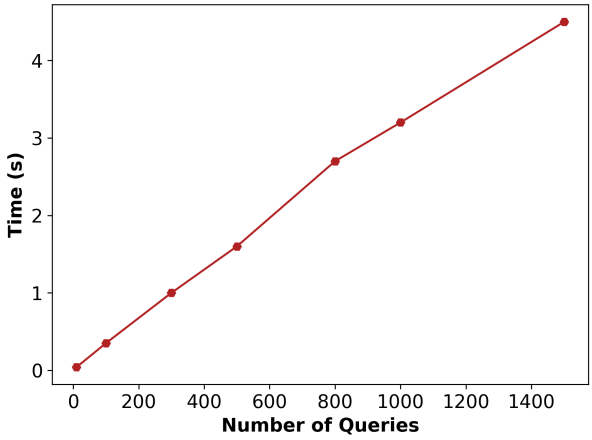
# Result Size (Kb)	Time (ms)
9	55
43	226
87	437
406	2397
733	4440
928	5582
1486	8901

Table 6.3: Varied result size

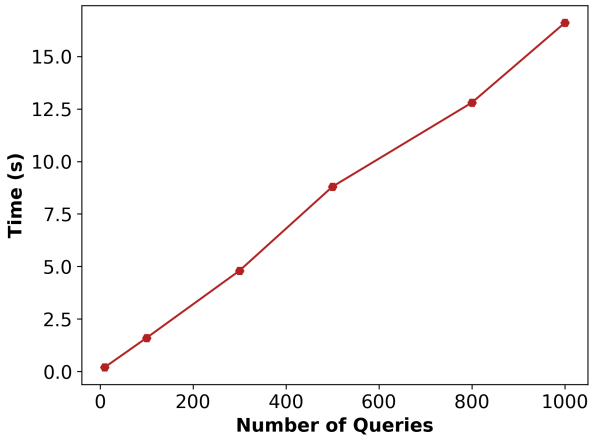
6.4.4 Prioritization Techniques: Simulation Results

In this simulation evaluation, we study the performance of our proposed algorithms against a set of performance metrics including total value of notifications that have been delivered to subscribers, average user satisfaction, and user fairness. For simplicity, we assume all notifications have the same size of 88 bytes. The broker is constrained by its outbound bandwidth, which is the upper bound on the total size of notifications that can be delivered to subscribers in our time unit. This results in the constraint on the maximum number of notifications that can be delivered in one time unit. We use β to denote the total number of notifications each broker can disseminate to subscribers in one time unit.

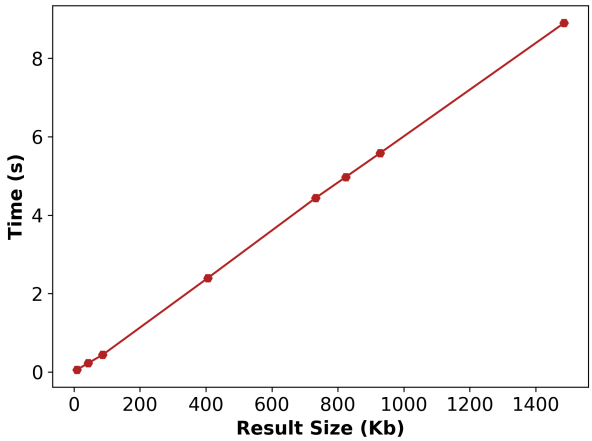
First, we measure the performances of all proposed prioritization algorithms on the two metrics: total value of delivered notifications and the average user satisfaction as shown in Figure 6.9 a) and Figure 6.9 b), respectively. The red line in Figure 6.9 a), represents the upper bound on total value of notifications to subscribers - this is an ideal hypothetical setting where there are no delays incurred by brokers in retrieving results from the BDMS and there are no constraints on broker outbound bandwidth (β equals infinity). As shown in Figure



(a) Result size of 88 b



(b) Result size of 9 Kb

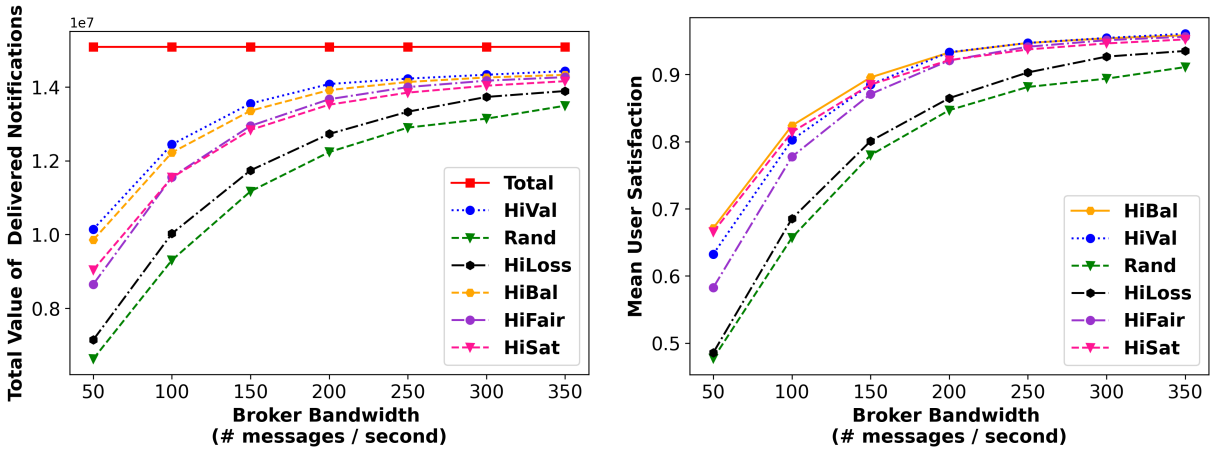


(c) 1 Query

Figure 6.8: Prototype Measurements

6.9 a) and b), when we increase β value, the total number of messages a broker can deliver in a time unit, the total value of delivered notifications and the average user satisfaction also increase. The HiVal algorithm gives the highest value of delivered notifications. The HiBal algorithm gives good performance on both metrics: high delivered notification value and high average user satisfaction. We conclude that, HiVal prioritization technique should be chosen if achieving the highest total value of delivered notifications is the goal; however, HiBal is a better choice if both system-wide notification value and overall user satisfaction are both important.

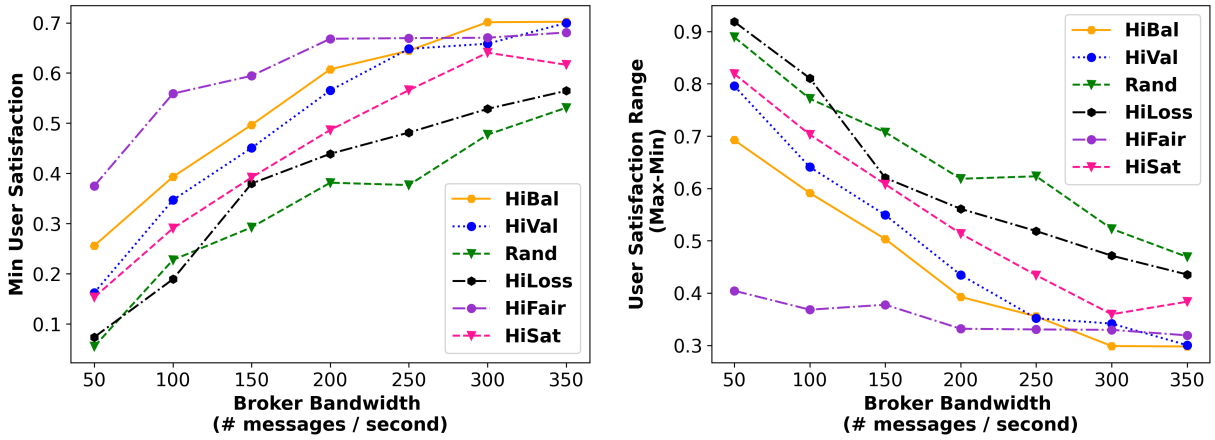
Second, we evaluate the proposed set of prioritization algorithms on the user fairness perfor-



(a) (b)

Figure 6.9

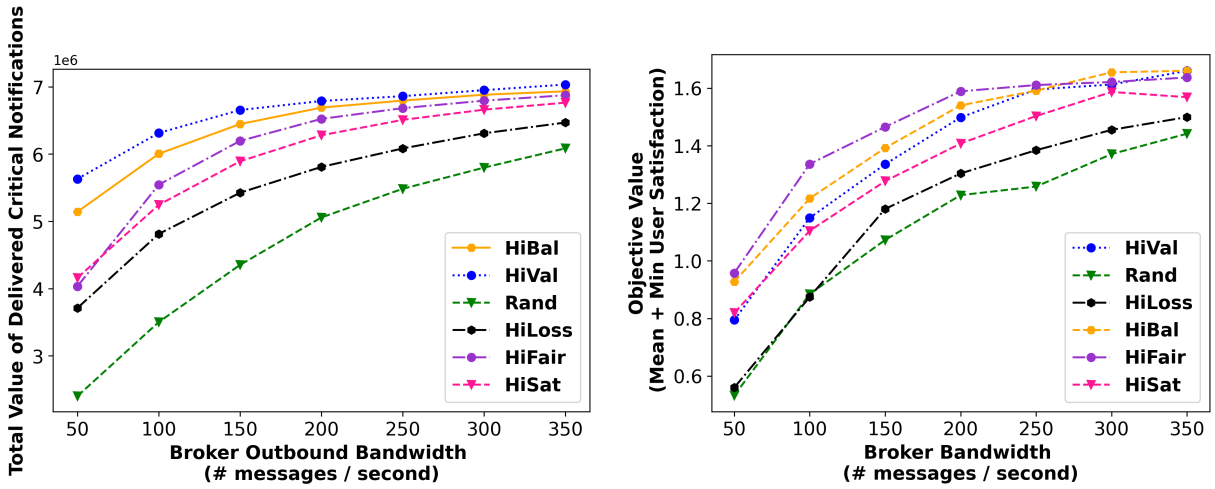
mance metric. We present several fairness indicators such as: minimum user satisfaction, gap between maximum user satisfaction and minimum user satisfaction as shown in Figures 6.10 a) and 6.10 b), respectively. We can see that HiFair algorithm leads to the highest minimum user satisfaction and the smallest gap between maximum user satisfaction and minimum user satisfaction. Again, HiBal algorithm is the second best for user fairness. We conclude that HiFair prioritization technique should be chosen to prioritize fairness among end-users. From results shown in Figure 6.9, 6.10, we recommend HiBal prioritization technique for both user satisfaction and fairness among end-users.



(a) (b)

Figure 6.10

Finally, we examine the proposed algorithms against other performance metrics including the total value of delivered notifications from critical channels, and the objective function value which is the sum of average user satisfaction and minimum user satisfaction, as shown in Figure 6.11 a) and b) respectively. Again, HiVal algorithm delivers the maximum value of notifications from critical channels. We recommend HiVal prioritization technique when the system wants to deliver maximum value of notifications from critical channels, e.g., in disaster scenarios. We recommend the HiBal prioritization technique when the system aims to deliver highest total value of notifications from critical channels while enabling fairness among end-users (e.g., to make sure all users receives fair value of delivered notifications from critical channels).



(a) (b)

Figure 6.11

6.5 Conclusion

In this chapter, we propose and demonstrate the effectiveness of various prioritization techniques for the notification prioritization problem in BDPS systems. We aim to maximize the total benefit that subscribers receive from the system ,i.e. to maximize total notification

value received by subscribers, maximize overall user satisfaction, and ensure fairness among users when system experiences unexpectedly high workloads and fails to deliver all notifications to all subscribed end-users due to limited resources. We propose a model for evaluating the value of notifications to end-users taking into account channel characteristics and user preferences. We also define metrics for user satisfaction and user fairness evaluation. Overall, we aim to balance the overall satisfaction of all users and fairness among users within the system.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we studied different challenges and introduced techniques, and services for enhancing the scalability, reliability and efficiency of BDPS systems under dynamic conditions.

In Chapter 4, we developed a multistage adaptive load balancing framework to mitigate the dynamically skewed load distribution among brokers. Our multistage load balancing framework consisted of three phases: i) *Initial Subscriber Placement*, which explored different techniques to first assign subscribers to brokers, ii) *Dynamic Migration* which tried to achieve moderately skewed load distributions among brokers by migrating subscribers from highly loaded brokers to lightly loaded brokers, and iii) *Shuffle* which redistributed the entire set of subscribers among all brokers to fix extremely skewed load distributions.

In Chapter 5, we designed and implemented REAPS (REliable Active Publish Subscribe) - a primary-backup fault tolerance framework to handle different classes of broker failures, including randomized failures and geo-correlated failures (e.g., in natural disasters). REAPS

exploited subscription similarity among brokers and applied a quasi-active state replication for low overhead, enabling fast recovery and delivery guarantee of notification service.

Chapter 6 developed notification prioritization techniques for efficient notification dissemination during periods of unanticipated high workload in the system. We developed a model for evaluating the value of notifications to end-users based on channel characteristics and user preferences. In addition, we defined metrics for evaluating user satisfaction and user fairness. Overall, we developed prioritization and delivery scheduling policies to maximize the total value of delivered notifications to subscribers, total user satisfaction, and ensure fairness among users when the system cannot deliver all notifications to subscribers before their deadlines.

7.2 Future Work

This thesis leads to a number of interesting future research directions:

- **Applying prediction models in the load balancing work in Chapter 4:** In this work, we developed load balancing techniques based on the current system states, such as subscriptions from subscribers, subscribers-to-brokers mappings, and notification data rates for each subscription. The next step in this direction could be to develop prediction models for the overall system workloads and load distributions, based on the subscription patterns from end-users, publication patterns from publishers, notification patterns, and subscriber movements (which may affect the mapping of subscribers to brokers, e.g., in geo-location-based broker assignments to subscribers). These can be leveraged to design more sophisticated load balancing techniques to minimize redundant subscriber migrations caused by instant load imbalances.

- **Exploiting notification correlation, examining other prioritization techniques**: Currently, we evaluate the value of notifications to end-users individually for the notification prioritization work in Chapter 6. We assume that the total value of delivered notifications to subscribers equals the sum of all delivered notification values. We do not look at the correlation among delivered notifications to a subscriber. Furthermore, in this work, we designed notification prioritization techniques by leveraging the current system states at the broker level, such as the notification value state and the notification queue state. This work could be extended by exploring and examining other techniques such as Lyapunov-based control techniques, which perform optimizations across time units.

- **Incorporating all developed schemes and services in one integrated solution:** In this thesis, we have addressed each research problem separately. Given these load balancing, fault tolerance, and notification prioritization techniques, the next step would be to integrate them into one comprehensive solution for a scalable, reliable, and efficient BDPS system. The integrated solution should determine appropriate techniques to trigger as needed. Security issues such as DDOS attacks are interesting to consider, e.g., overwhelming the system and creating unfairness.

Bibliography

[1] Asterixdb. `https://asterixdb.apache.org`.

[2] Big active data @ uci. `http:\/\/asterix.ics.uci.edu`.

[3] Big active data @ ucr. `http://www.cs.ucr.edu/~tsotras/big-active-data/index.htm`.

[4] Zotalert. `https://www.oit.uci.edu/zotalert/`.

[5] H. M. Ali and Z. S. Alwan. *Car accident detection and notification system using smartphone*. LAP LAMBERT Academic Publishing Saarbrucken, 2017.

[6] 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)*, pages 1345–1356. IEEE, 2020.

[7] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613, 2018.

[8] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[9] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *DIS, Universita di Roma La Sapienza, Tech. Rep*, 5, 2005.

[10] D. Barbará. The characterization of continuous queries. *International Journal of Cooperative Information Systems*, 8(04):295–323, 1999.

[11] 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*, pages 481–482. IEEE Computer Society, 1990.

[12] A. Basak, K. Venkataraman, R. Murphy, and M. Singh. *Stream Analytics with Microsoft Azure: Real-time data processing for quick insights using Azure Stream Analytics*. Packt Publishing Ltd, 2017.

[13] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

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

[15] 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*, pages 321–343. Springer, 1993.

[16] 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*, pages 181–186, 2016.

[17] 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)*, 19(3):332–383, 2001.

[18] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[19] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable xml data dissemination. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 826–837. Elsevier, 2002.

[20] C. Chen, H.-A. Jacobsen, and R. Vitenberg. Algorithms based on divide and conquer for topic-based publish/subscribe overlay design. *IEEE/ACM Transactions on Networking*, 24(1):422–436, 2014.

[21] 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*, pages 105–116, 2016.

[22] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, 2000.

[23] A. K. Y. Cheung and H.-A. Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Transactions on Computer Systems (TOCS)*, 28(4):1–55, 2010.

[24] A. K. Y. Cheung and H.-A. Jacobsen. Green resource allocation algorithms for publish/subscribe systems. In *2011 31st International Conference on Distributed Computing Systems*, pages 812–823. IEEE, 2011.

[25] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[26] H. Cole-Lewis and T. Kershaw. Text messaging as a tool for behavior change in disease prevention and management. *Epidemiologic reviews*, 32(1):56–69, 2010.

[27] D. Dedousis, N. Zacheilas, and V. Kalogeraki. On the fly load balancing to address hot topics in topic-based pub/sub systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 76–86. IEEE, 2018.

[28] M. Deshpande, K. Kim, B. Hore, S. Mehrotra, and N. Venkatasubramanian. Recrew: A reliable flash-dissemination system. *IEEE Transactions on Computers*, 62(7):1432–1446, 2012.

[29] N. Do and N. Venkatasubramanian. Rich content sharing in mobile systems using multiple wireless networks. In *Proceedings of the 9th Middleware Doctoral Symposium of the 13th ACM/IFIP/USENIX International Middleware Conference*, pages 1–5, 2012.

[30] N. M. Do, C.-H. Hsu, and N. Venkatasubramanian. Hybcast: Rich content dissemination in hybrid cellular and 802.11 ad hoc networks. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 352–361. IEEE, 2012.

[31] F. T. El-Hassan and D. Ionescu. Design and implementation of a hardware versatile publish-subscribe architecture for the internet of things. *IEEE Access*, 6:31872–31890, 2018.

[32] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[33] P. Felber. The corba object group service: A service approach to object groups in corba. 1998.

[34] J. Gascon-Samson, F.-P. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 486–496. IEEE, 2015.

[35] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *IEEE INFOCOM 2004*, volume 4, pages 2253–2262. IEEE, 2004.

[36] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[37] 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*, pages 254–273. Springer, 2004.

[38] G. Gupta and M. Younis. Load-balanced clustering of wireless sensor networks. In *IEEE International Conference on Communications, 2003. ICC'03.*, volume 3, pages 1848–1852. IEEE, 2003.

[39] W. Han, S. Ada, R. Sharman, and H. R. Rao. Campus emergency notification systems. *Mis Quarterly*, 39(4):909–930, 2015.

[40] E. S. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed systems*, 5(2):113–120, 1994.

[41] S. Huq, Z. Shafiq, S. Ghosh, A. Khakpour, and H. Bedi. Distributed load balancing in key-value networked caches. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 583–593. IEEE, 2017.

[42] 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 big active data. *Proceedings of the VLDB Endowment*, 10(12):1941–1944, 2017.

[43] 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*, 29:1337–1364, 2020.

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

[45] H.-A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The padres publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*, pages 164–205. IGI Global, 2010.

[46] 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*, pages 52–59. IEEE, 2008.

[47] H. Jafarpour, S. Mehrotra, and N. Venkatasubramanian. Dynamic load balancing for cluster-based publish/subscribe system. In *2009 Ninth Annual International Symposium on Applications and the Internet*, pages 57–63. IEEE, 2009.

[48] S. Ji, C. Ye, J. Wei, and H.-A. Jacobsen. Merc: Match at edge and route intra–cluster for content-based publish/subscribe systems. In *Proceedings of the 16th Annual Middleware Conference*, pages 13–24, 2015.

[49] 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*, pages 41–50. IEEE, 2009.

[50] 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, 2009. ICST.

[51] A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, volume 2, pages 243–250. IEEE, 1994.

[52] 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*, pages 169–190. Springer, 2010.

[53] K. Kim, S. Mehrotra, and N. Venkatasubramanian. Efficient and reliable application layer multicast for flash dissemination. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2571–2582, 2013.

[54] K. Kim, Y. Zhao, and N. Venkatasubramanian. Gsford: Towards a reliable geo-social notification system. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 267–272. IEEE, 2012.

[55] 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*, pages 27–38, 2013.

[56] S. Krishnan and J. L. U. Gonzalez. Google cloud pub/sub. In *Building Your Next Big Thing with Google Cloud Platform*, pages 277–292. Springer, 2015.

[57] E. D. Kuligowski, E. D. Kuligowski, and A. Kimball. *Alerting under imminent threat: Guidance on alerts issued by outdoor siren and short message alerting systems.* US Department of Commerce, National Institute of Standards and Technology, 2018.

[58] J. Li, Z. Ning, B. Jedari, F. Xia, I. Lee, and A. Tolba. Geo-social distance-based data dissemination for socially aware networking. *IEEE Access*, 4:1444–1453, 2016.

[59] A. Lima and M. Musolesi. Spatial dissemination metrics for location-based social networks. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 972–979, 2012.

[60] C. P. Low, C. Fang, J. M. Ng, and Y. H. Ang. Efficient load-balanced clustering algorithms for wireless sensor networks. *Computer Communications*, 31(4):750–759, 2008.

[61] M. Ma, Z. Wang, K. Yi, J. Liu, and L. Sun. Joint request balancing and content aggregation in crowdsourced cdn. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1178–1188. IEEE, 2017.

[62] S. T. Maguluri, R. Srikant, and L. Ying. Stochastic models of load balancing and scheduling in cloud computing clusters. In *2012 Proceedings IEEE Infocom*, pages 702–710. IEEE, 2012.

[63] A. Mehrotra and M. Musolesi. Intelligent notification systems: A survey of the state of the art and research challenges. *arXiv preprint arXiv:1711.10171*, 2017.

[64] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. *GeoInformatica*, 9(4):343–365, 2005.

[65] H. Nguyen, M. Uddin, and N. Venkatasubramanian. Reaps: Quasi-active fault tolerance for big data publish-subscribe systems. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 368–375. IEEE, 2021.

[66] 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*, pages 43–54, 2019.

[67] I. H. Osman. Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches. *Operations-Research-Spektrum*, 17(4):211–225, 1995.

[68] J. Qi, R. Zhang, C. S. Jensen, K. Ramamohanarao, and J. He. Continuous spatial query processing: a survey of safe region based techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[69] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *International Workshop on Peer-to-Peer Systems*, pages 68–79. Springer, 2003.

[70] G. T. Ross and R. M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming*, 8(1):91–103, 1975.

[71] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[72] 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*, pages 30–43. Springer, 2001.

[73] 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*, pages 93–104, 2016.

[74] P. Salehi, K. Zhang, and H.-A. Jacobsen. Popsub: Improving resource utilization in distributed content-based publish/subscribe systems. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 88–99, 2017.

[75] 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*, pages 271–291. Springer, 2012.

[76] H. Shen. Content-based publish/subscribe systems. In *Handbook of peer-to-peer networking*, pages 1333–1366. Springer, 2010.

[77] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. *arXiv preprint cs/9810019*, 1998.

[78] S. Tang, B. He, C. Yu, Y. Li, and K. Li. A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[79] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *Acm Sigmod Record*, 21(2):321–330, 1992.

[80] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[81] V. Turau and G. Siegemund. Scalable routing for topic-based publish/subscribe systems under fluctuations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1608–1617. IEEE, 2017.

[82] M. Y. S. Uddin, V. Setty, Y. Zhao, R. Vitenberg, and N. Venkatasubramanian. Richnote: Adaptive selection and delivery of rich media notifications to mobile users. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 159–168. IEEE, 2016.

[83] 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)*, pages 696–705. IEEE, 2018.

[84] G. Van Dongen and D. Van den Poel. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1845–1858, 2020.

[85] D. Wald, K. Lin, L. Turner, and N. Bekiri. Us geological survey's shakecast system: A cloud-based future. In *Proceedings of the Tenth US National Conference on Earthquake Engineering (10NCEE)*, 2014.

[86] X. Wang. *Activating Big Data at Scale*. University of California, Irvine, 2020.

[87] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[88] F. Xia, A. M. Ahmed, L. T. Yang, and Z. Luo. Community-based event dissemination with optimal load balancing. *IEEE Transactions on Computers*, 64(7):1857–1869, 2014.

[89] F. Xia, L. Liu, J. Li, J. Ma, and A. V. Vasilakos. Socially aware networking: A survey. *IEEE Systems Journal*, 9(3):904–921, 2013.

[90] B. Xing, M. Deshpande, S. Mehrotra, and N. Venkatasubramanian. Gateway designation for timely communications in instant mesh networks. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 564–569. IEEE, 2010.

[91] B. Xing, S. Mehrotra, and N. Venkatasubramanian. Disruption-tolerant spatial dissemination. In *2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 1–9. IEEE, 2010.

[92] B. Xing, K. Seada, and N. Venkatasubramanian. An experimental study on wi-fi ad-hoc mode for mobile device-to-device video delivery. In *IEEE INFOCOM Workshops 2009*, pages 1–6. IEEE, 2009.

[93] 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*, pages 800–811. IEEE, 2011.

[94] N. Zacheilas, N. Zygouras, N. Panagiotou, V. Kalogeraki, and D. Gunopulos. Dynamic load balancing techniques for distributed complex event processing systems. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 174–188. Springer, 2016.

[95] 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*, pages 75–86, 2013.

[96] Y. Zhao and W. Song. Survey on social-aware data dissemination over mobile wireless networks. *IEEE Access*, 5:6049–6059, 2017.

[97] Y. Zhou, A. Salehi, and K. Aberer. Scalable delivery of stream query result. In *Proceedings of 35th International Conference on Very Large Data Bases (VLDB 2009)*, number CONF. VLDB, 2009.

[98] Y. Zhu and Y. Hu. Efficient, proximity-aware load balancing for dht-based p2p systems. *IEEE Transactions on parallel and distributed systems*, 16(4):349–361, 2005.

[99] B. Zolfaghari, G. Srivastava, S. Roy, H. R. Nemati, F. Afghah, T. Koshiba, A. Razi, K. Bibak, P. Mitra, and B. K. Rai. Content delivery networks: State of the art, trends, and future roadmap. *ACM Computing Surveys (CSUR)*, 53(2):1–34, 2020.

Appendix A

```
/****** Three Datasets ******/

//create datatypes

create type UserLocationType if not exists as open{
recordId: uuid,
latitude: double,
longitude: double,
userName: string,
timeoffset: float,
timestamp: datetime
};

create type UserLocationFeedType if not exists as open{
latitude: double,
longitude: double,
userName: string,
timeoffset: float,
```

```
timestamp: datetime
};

create type EmergencyReportType if not exists as open {
recordId: uuid,
severity: int,
impactZone: circle,
timeoffset: float,
timestamp: datetime,
duration: float,
message: string,
emergencyType: string,
userName: string
};

create type EmergencyReportFeedType if not exists as open {
severity: int,
impactZone: circle,
timeoffset: float,
timestamp: datetime,
duration: float,
message: string,
emergencyType: string,
userName: string
};
```

```
create type EmergencyShelterType if not exists as open {
name: string,
location: point
};


//create datasets
create dataset EmergencyReports(EmergencyReportType) primary key
recordId AUTOGENERATED;
create dataset UserLocations(UserLocationType) primary key recordId
AUTOGENERATED;
create dataset EmergencyShelters(EmergencyShelterType) primary key name;

create index gbTimestampReport on EmergencyReports(timestamp);
create index gbTimestampLocation on UserLocations(timestamp);
create index gbEmergencyTypeReport on EmergencyReports(emergencyType) type keyword;
create index locReports on EmergencyReports(impactZone) type RTREE;
create index locShelters on EmergencyShelters(location) type RTREE;

/****** Five Channels ******/

//create functions
//1) "All emergencies of type T"
create function recentEmergenciesOfType(emergencyType){
(select r as reports from
(select value r from EmergencyReports r where r.timestamp >
current_datetime() - day_time_duration("PT10S")) r
```

```
where r.emergencyType = emergencyType)
};
//2) "All emergencies at location L"
create function recentEmergenciesAtLocation(latitude, longitude){
(select r as reports from
(select value r from EmergencyReports r where r.timestamp >
current_datetime() - day_time_duration("PT20S")) r
where spatial_intersect(r.impactZone,create_point(latitude,longitude)))
};

//3) "All emergencies of type T at location L"
create function recentEmergenciesOfTypeAtLocation(emergencyType, latitude, longitude){
(select r as reports from
(select value r from EmergencyReports r where r.timestamp >
current_datetime() - day_time_duration("PT30S")) r
where r.emergencyType = emergencyType
and spatial_intersect(r.impactZone,create_point(latitude,longitude)))
};

//4) "All emergencies of type T at location L with shelters S"
create function
recentEmergenciesOfTypeAtLocationWithShelter(emergencyType, latitude, longitude)
{
(select r as reports, shelter as shelters
from (select value r from EmergencyReports r where r.timestamp
> current_datetime() - day_time_duration("PT60S")) r
```

```
let shelter = (select value shelter from EmergencyShelters shelter
  where spatial_intersect(r.impactZone, shelter.location))
where r.emergencyType = emergencyType
and spatial_intersect(r.impactZone,create_point(latitude,longitude)))
};

//5) "The impactZone and message for all emergencies intersecting user U"
create function recentEmergenciesNearMe(userName){
  (select r as reports
  from
  (select value r from EmergencyReports r where r.timestamp >
  current_datetime() - day_time_duration("PT10S")) r,
  (select value l from UserLocations l where l.timestamp >
  current_datetime() - day_time_duration("PT10S")) user
   where user.userName = userName
   and spatial_intersect(r.impactZone,create_point(user.latitude,user.longitude)))
};

//6) "The impactZone and message for all emergencies of type T intersecting user U"
create function recentEmergenciesOfTypeNearMe(emergencyType, userName){
  (select r as reports
  from
  (select value r from EmergencyReports r where r.timestamp >
  current_datetime() - day_time_duration("PT10S")) r,
  (select value l from UserLocations l where l.timestamp >
  current_datetime() - day_time_duration("PT10S")) user
where user.userName = userName
```

```
  and r.emergencyType = emergencyType
  and spatial_intersect(r.impactZone,create_point(user.latitude,user.longitude)))
};

//7) "The impactZone, message, and a list of Shelters for all
emergencies of type T intersecting user U"
create function recentEmergenciesOfTypeWithShelterNearMe(emergencyType, userName){
(select r as reports, shelter as shelters
from
  (select value r from EmergencyReports r where r.timestamp >
  current_datetime() - day_time_duration("PT10S")) r,
  (select value l from UserLocations l where l.timestamp >
  current_datetime() - day_time_duration("PT10S")) user
let shelter = (select value shelter from EmergencyShelters shelter
  where spatial_intersect(r.impactZone, shelter.location))
where user.userName = userName
  and r.emergencyType = emergencyType
  and spatial_intersect(r.impactZone,create_point(user.latitude,user.longitude)))
};


create repetitive channel recentEmergenciesOfTypeChannel using
recentEmergenciesOfType@1 period duration("PT10S");
create repetitive channel recentEmergenciesAtLocationChannel using
recentEmergenciesAtLocation@2 period duration("PT20S");
create repetitive channel recentEmergenciesOfTypeAtLocationChannel
using recentEmergenciesOfTypeAtLocation@3 period duration("PT30S");
```

```
create repetitive channel recentEmergenciesNearMeChannel using
recentEmergenciesNearMe@1 period duration("PT10S");

/****** Three Feeds ******/

//create feeds
create feed ReportFeed with
{
    "adapter-name": "socket_adapter",
"sockets": "promethium.ics.uci.edu:23231",
"address-type": "IP",
"type-name": "EmergencyReportFeedType",
"format": "adm"
};

create feed ShelterFeed with
{
    "adapter-name": "socket_adapter",
    "sockets": "promethium.ics.uci.edu:23232",
"address-type": "IP",
"type-name": "EmergencyShelterType",
"format": "adm"
};

create feed UserLocationFeed with
{
"adapter-name": "socket_adapter",
```

```
"sockets": "promethium.ics.uci.edu:23233",
"address-type": "IP",
"type-name": "UserLocationFeedType",
"format": "adm"
};

//connect feeds
connect feed ReportFeed to dataset EmergencyReports;
connect feed ShelterFeed to dataset EmergencyShelters;
connect feed UserLocationFeed to dataset UserLocations;

//start feeds
start feed ReportFeed;
start feed ShelterFeed;
start feed UserLocationFeed;
```