Sapienza University of Rome

Ph.D. program in Computer Engineering

XXIV Cycle - 2011

Ordering, Timeliness and Reliability for Publish/Subscribe Systems over WAN

Marco Platania

Sapienza University of Rome

Ph.D. program in Computer Engineering

XXIV Cycle - 2011

Marco Platania

Ordering, Timeliness and Reliability for Publish/Subscribe Systems over WAN

Thesis Committee

Prof. Roberto Baldoni (Advisor)
Prof. Silvio Salza

Reviewers

Prof. Hans-Arno Jacobsen
Prof. Peter Pietzuch

Author's address:
Marco Platania
Dipartimento di Ingegneria Informatica Automatica e Gestionale
"A. Ruberti"
Sapienza Università di Roma
Via Ariosto 25, 00185 Roma, Italy
e-mail: platania@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼platania

Contents

Chapter 1

Introduction

1.1 Context and motivations

The geo-political, sociological and financial changes induced by globalization in the last few years, are creating the need for a connected world where the right information is always available at the right time. As such, with the increasing use of the Internet and the improvement in hardware technology, most of the applications previously deployed in "closed" environment are federating into geographically-distributed systems. In the future, no system is expected to be isolated; every system will be composed by the interconnection of independent systems that need to share information, i.e., it will be a *System-of-Systems (SoS)*. Example of SoS are *Large scale Complex Critical Infrastructures (LCCIs)*, such as power grids, transport infrastructures (airports and seaports), financial infrastructures, next generation intelligence platforms, to cite a few. LCCIs play a fundamental role into several human activities and have a strong economic and social impact. Hence, the consequences of an outage can be catastrophic in terms of efficiency, economical losses, consumer dissatisfaction and even indirect harm to people. As a consequence, these systems are being confronted with new scenarios and operational requirements posing unprecedented challenges with respect to: (i) the scale at which they need to operate; (ii) the ubiquity and multi-modality of data access they need to provide; (iii) the *Quality of Service (QoS)* level they need to deliver.

The proliferation of large scale applications and the process of federating different systems in SoS, pose particular attention on the middleware used to connect system nodes and to distribute data possibly from multiple sources to all interested destinations. In the last few years, the *publish/subscribe*

paradigm is becoming attractive for anonymous and asynchronous information dissemination. Specifically, this middleware is characterized by three kinds of entities, as shown in Figure 1.1: (i) *publishers*, i.e, the producers of information, (ii) *subscribers*, i.e., the consumers of information, and (iii) an event notification service composed by brokers that convey the information from publishers to subscribers.

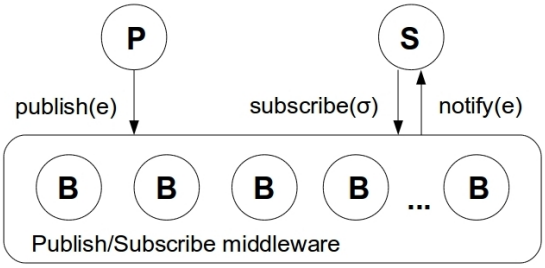


Figure 1.1: Architecture of a publish/subscribe middleware. The interaction between a publisher (P) and a subscriber (S) is mediated by a network of brokers (B).

The communication takes the form of *event*, where an event is any occurrence of something of interest [88]. The main benefit of the publish/subscribe paradigm lies on its intrinsic time, space and synchronization decoupling properties [52]: publishers and subscribers do not need to know each other, or to be online at the same time, or to operate in a synchronous way. These properties are suitable to satisfy scalability requirements exhibited by large scale systems, so to make the publish/subscribe paradigm attractive for the design of innovative critical infrastructure. A concrete example is represented by the novel Air Traffic Management (ATM) framework implemented in Europe by EUROCONTROL in the context of the SESAR European Project [98], where the *System Wide Information Management* (SWIM) middleware is used to federate geographically sparse ATM entities, that act both as producers and consumers of information (a complete description of the SESAR architecture will be provided in Section 1.2.4).

The information sharing in a SoS is conveyed by wide-area channels, due to its geographical extension. As such, the communication may be affected by the unpredictable behavior of the network, where messages can be dropped or delayed due to possible link failures or congestions. However, due to their critical nature, LCCIs require that the middleware used to route data from the source to all intended destinations provides an adequate QoS level. In fact, for this kind of infrastructures, a message loss or a late delivery can compromise

the mission of the overall system, leading to catastrophic consequences as described above. Hence, a publish/subscribe middleware has to provide a set of non-functional aspects, in order to support the implementation of the novel generation of LCCIs. This set is orthogonal to the middleware core functionalities, and includes QoS properties that we group in three categories: reliability, ordering and timeliness.

Reliability properties define which kind of guarantees are provided in the delivery of each single event to its intended destinations. We can distinguish between two different properties:

- *Best-effort delivery:* the middleware will provide the event routing service without any specific guarantee on the delivery; from this point of view, receivers can expect to miss some events that will not be delivered due to unexpected causes (e.g. message losses in the underlying transport layer); publish/subscribe services providing best-effort delivery are usually designed to offer the best performance (obtained by avoiding as much as possible any protocol overhead); despite the lack of specific mechanisms to enforce stronger delivery properties, such systems usually behave in a good way as long as they are deployed on top of a fairly reliable communication substrate.

- *Reliable delivery:* the middleware will guarantee the delivery of events to all their intended destinations despite possible network losses or other unexpected events; this property is usually enforced at the event routing level through several different techniques such as retransmission, broker replication, multi-path delivery, etc. The overhead caused by the property enforcement can a have a non-negligible impact on the overall performance.

Note that the enforcement of a reliable delivery property implies a strict definition in the publish/subscribe middleware of the set of events that must be delivered to each destination. This becomes a non-trivial issue as soon as one considers that destination processes can possibly change at runtime their subscriptions and that the interactions taking place between event producers and consumers are completely asynchronous and decoupled with the purpose of improving system scalability.

Ordering properties provide a mean to define a specific order that must be enforced by the publish/subscribe service when delivering events to consumers. Several different, and partially orthogonal ordering properties can be considered:

- *Per-source FIFO order:* this is the simplest form of order that can be considered as it forces the middleware to deliver events published by a

same producer in the same order as they were produced (i.e. with a FIFO semantic); as a consequence, the delivery of events generated by distinct sources can experience different interleaving on distinct destinations.

- *Causal order:* by enforcing this property, the middleware guarantees that if a process publishes an event e' after an event e has been delivered to it, then the publish/subscribe service will not deliver to a second process the events in the order e', e; this property proves particularly useful in those contexts where processes act both as producers and consumers and where maintaining the causal relationship among multiple events is fundamental for the application correctness.

- *Total order:* by enforcing this property, the middleware guarantees that any two destinations that receive a same set of events will receive those events exactly in the same order; this order does not necessarily have a connection with the real publishing time instants of these events, i.e. two events e and e' published at time t and t', with $t < t'$ can be delivered either in the order t, t' or t', t, but not a mix of the two.

- *Real-time order:* this property closely resembles the total order property but also require deliveries to be executed in the same order as events have been produced (hence the *real-time* attribute); the enforcement of this property can be usually guaranteed only with a tolerance due to the impossibility to perfectly synchronize event sources and thus always correctly order the publication time of concurrent events.

The real-time order property is clearly the most comprehensive as it subsumes all the preceding ones. Causal and total order are orthogonal and can be enforced together, if needed.

Timeliness expresses the ability of the publish/subscribe middleware to provide the expected service within known time bounds [15]. This aspect proves crucial in many mission-critical applications, and it directly impacts the time needed to route an event from its publication point up to all its intended destinations.

- *Timely delivery:* there exists a time interval Δ such that, given any event e delivered at a node at time t, e has been published at a time t' where $t - \Delta \leq t' < t$.

In the remainder of this Chapter, we present several case studies of real applications, by highlighting which of the properties defined above need to be satisfied by the publish/subscribe middleware used in those specific contexts (Section 1.2). Section 1.4 describes how QoS is addressed in publish/subscribe

systems and the current solutions present in literature to ensure ordering, timeliness and reliability. From this assessment, we analyze the gap between the requirements of the applications (summarized in Section 1.3) and the QoS level provided by current middleware solutions (Section 1.4), highlighting which aspects need to be considered for event dissemination over WAN and which challenges they pose (Section 1.5). This leads to the contribution of the thesis, discussed in Section 1.6, that consists in designing a framework that can be posed on top of a generic publish/subscribe middleware to address some of the presented QoS properties, specifically total ordering, reliable and timely delivery.

1.2 Case studies

In this Section we discuss different application domains highlighting the QoS requirements that a publish/subscribe middleware has to satisfy in order to support those applications.

1.2.1 Collaborative Security for Financial Critical Infrastructures Protection

Financial institutions are increasingly exposed to a variety of security related risks, such as massive and coordinated cyber attacks [4, 5] aiming at capturing high value (or, otherwise, sensitive) information, or disrupting the service operation for various purposes. Single financial institutions use local tools to protect themselves from those attacks (e.g. intrusion detection systems, firewalls); these tools verify whether there exists some host that performs suspicious activities within certain time windows. However, due to the complexity of today's attacks, such kind of defense results inadequate. A more large view of what is happening at all financial institution sites is required, that could be obtained by *collaboratively* sharing and correlating the information coming from them, thus improving chances of identifying low volume activities which would have gone undetected if individual institutions were exclusively relying on their local protection systems [81].

Figure 1.2 illustrates the scenario of collaborative protection of financial critical infrastructures against inter-domain stealthy SYN port scan attacks[1]. The attack is a form of port scan that aims at uncovering the status of certain TCP ports without being traced by application level loggers. It is carried

[1]This scenario has been widely investigated in the context of the EU project CoMiFin [38]

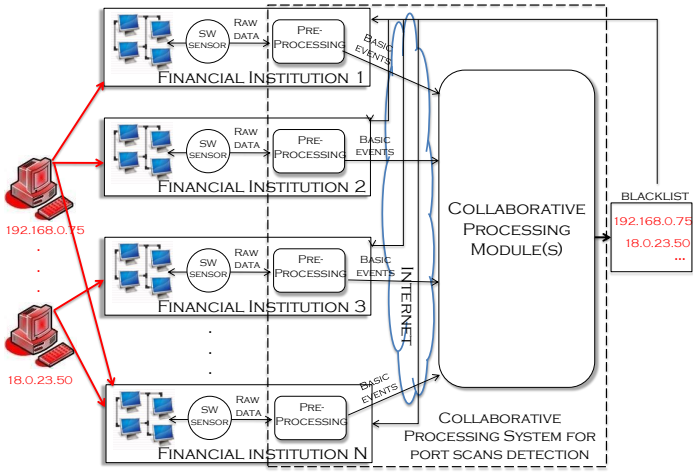


Figure 1.2: Collaborative processing system for port scan detection in financial critical infrastructures.

out by probing a few ports of interest at different financial institutions, so as to circumvent configured thresholds of local protection systems, and delaying those probes in order to also bypass local time window controls [6, 7].

A scanner targeting different financial institutions probes ports on different targets by following a well-known data pattern. For instance, a common pattern in an inter-domain stealthy port scan is to initiate so-called *incomplete connections*; that is, TCP connections for which the three-way handshaking consists of the following ordered sequence "→" of packets: SYN → SYN-ACK → RST (or nothing after a timeout is expired).

Owing to this scenario, a possible collaborative processing system can be built such that it takes in input different basic events representing part of the traffic sniffed from financial sites' networks. It is worth noticing that basic events are obtained through specific pre-processing activities carried out locally by financial sites (see Figure 1.2). These activities allow the sites to control the data flow to be injected into the collaborative processing system: pre-filtering and possibly sensitive data anonymization operations are thus performed in this phase.

Basic events flow from multiple financial sources to one or more modules for collaborative processing purposes. These modules verify the presence of the earlier mentioned data pattern and detect whether that pattern is "frequently" discovered from all the sites (i.e., the processing modules verify if the total number of collaboratively detected incomplete connections exceeds

a pre-defined threshold). At the end of the processing, if a high number of malicious activities originated by specific IP addresses are observed, a blacklist containing those addresses is produced and disseminated to all the sites of the system (see Figure 1.2).

In the design of such an architecture, it clearly emerges the crucial role played by the middleware. It is used to convey both a large volume of basic events to the collaborative processing system, and the results of the processing to all the sites interested in receiving the produced blacklist.

However, in order to be effective, the middleware has to guarantee that specific QoS properties are satisfied in order to not compromise the port scan detection capabilities of the overall collaborative processing system.

Required middleware properties. For an accurate detection of inter-domain stealthy port scans, data dissemination should provide the following non-functional aspects:

• *reliable delivery* of both the basic events to the collaborative processing modules and the produced blacklist to financial sites of the system. Referring to the above data pattern, if the SYN-ACK packet is lost in the communication between the sources and processing modules, the pattern cannot be correctly detected. The pattern will then not contribute to the computation of the earlier discussed pre-defined threshold, thus augmenting the number of false negatives (i.e., real scans that are not detected) and decreasing the detection accuracy of the system.
• *per source FIFO order* of the basic events. Referring to the above data pattern, if some of those packets are delivered out of order at the level of single source, the processing modules will not recognize the correct sequence. This entails that even if the data pattern has been carried out by some malicious IP addresses, it cannot be detected and used in the general threshold computation. Similarly to the previous point, this may lead to augment the number of false negatives and, then, to decrease the port scan detection accuracy of the system.
• *timely delivery* of both the basic events and produced blacklist. Typically, these types of attacks are characterized by very low execution and propagation times, in the order of a few seconds depending on the number of target hosts and ports on those hosts. An effective processing system should then detect scanner IP addresses and disseminate the results to interested sites within that application time bound. This allows the sites to exploit the intelligence produced by the collaborative system and to timely take proper countermeasures. In doing so, the publish/subscribe middleware has to guarantee timeliness on both the delivery of basic events and the dissemination of produced results.

In particular, we claim that in the inter-domain stealthy port scan the final result of the collaborative computation should be delivered within a time interval which is comparable to the application time bound (i.e., within a few seconds).

1.2.2 Algorithmic Trading for Stock Market

The huge improvement in hardware and communication technology led the stock market to extend its services from physical locations, where buyers and sellers met and negotiated, toward a virtual market place (NASDAQ, NYSE Arca and Globex, to cite a few) that exploits electronic media, where traders can transact from remote locations. The increase of the electronic trading brougth several benefits, such as reduced cost of transactions, greater liquidity and competition (allowing different companies to trade with other ones) and increased price transparency [95].

On the contrary, the virtual market place poses unprecedented challenges in terms of data flows (up to 1 million updates per second) and short-term trading decisions (up to thousandths of a second). These problems are further exacerbated by the complexity of the operations. As an example, consider the following indication about when to buy or sell a stock: *"When the price of IBM is 0.5% higher than its average price in the last 30 seconds, buy 10000 shares of Microsoft every 3 seconds unless the average price drops back below the same threshold"* [90]. Such an indication requires a careful market analysis of the last IBM and Microsoft's share prices and a consequent decision whether to buy or not these shares. The complexity of operations and the strict timeliness constraints imposed by the electronic market, led to the use of sophisticated algorithms that process event streams, make complicated calculations and take intelligent decisions in response to changing conditions reflected in those events. This strategy, called *algorithmic trading*, is becoming more and more a fundamental component of the virtual market place: all the world's top-tier firms, including JP Morgan, Deutsche Bank, and ABN Amro, as well as buy-side hedge funds, such as Aspect Capital, are applying algorithmic trading.

A typical virtual market place is depicted in Figure 1.3. Each firm in the electronic market has its Trading Engine System: it is composed by several engines that take in input client orders stored in the Order Management System. Trading engines can obtain market data directly from the Exchange or from the Real Time Market Data. The Market Data sends information such as the current pricing and the number of contracts, and are provided in real time by the Exchange. Examples of the Market Data are Reuters, Bloomberg and Wombat. In addition, the Trading Engine System can always purchase historical market data directly from the market's Exchange. Trading engines

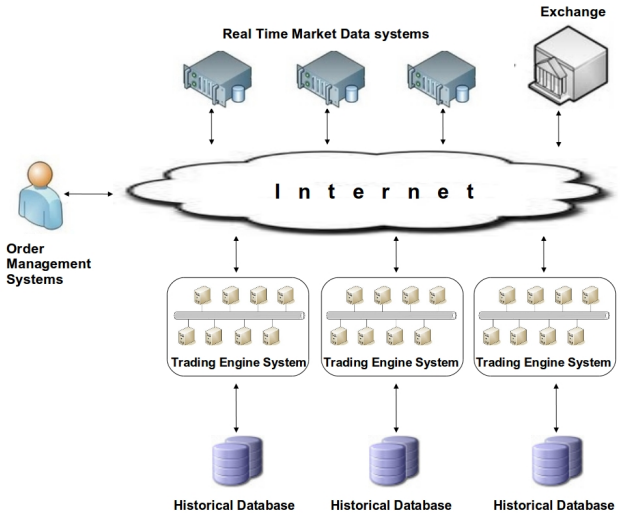


Figure 1.3: Virtual market place where the interaction between buyers and sellers occurs over the Internet.

execute complex queries on the data according to the instruction provided by clients. In addition, trading engines can be organized in a hierarchical manner, with some of them that operate on raw data and produce more complex events that will be analyzed by engines at a higher level in the hierarchy. The outcome of the executions and the order status is sent back to the Order Management System. Finally, raw and/or summarized historical data can also be stored on the Historical Database and utilized for future strategy decision.

The communication among the electronic market components flows through the Internet. Because of the strict requirements imposed by stock market applications, the publish/subscribe middleware plays a key role in the implementation of such system. In particular, it is required to convey a large volume of raw events from the Order Management System and the Exchange to the Trading Engine System in a reliable and timely fashion, in order to let trading engines properly apply their strategy.

Required middleware properties. The publish/subscribe middleware for algorithmic trading in stock market has to provide the following QoS properties:

• *reliable delivery:* raw data produced by the Exchange or the Real Time Market Data and complex events generated by trading engines, as well as client orders, outcome of operations and order status feedback, must be reliably delivered to all interested destinations. Indeed, the loss of a message can

compromise the accuracy of the applied strategy. As an example, consider the complex query described above, with a trading algorithm that has to buy 10000 Microsoft's shares when the price of IBM is 0.5% higher than its average price in the last 30 seconds. If the event *"the price of IBM is 0.5% higher than its average price"* occurred in the last 30 seconds is lost, the algorithm does not buy the Microsoft's shares, preventing the client from a possible profit.
• *timely delivery:* it is a strict requirement of the stock market applications, because the delay in the delivery of an event can lead the Trading Engine System to take a wrong decision. Consider, again, the previous example: if the event *"the price of IBM is 0.5% higher than its average price"* occurred in the last 30 seconds is notified to trading engines with a delay that exceedes the value Δ imposed by the application, then the algorithm might not buy the Microsoft's shares, preventing, even in this case, the client from a possible profit. In addition, the timeliness in the information delivery must be ensured not only for raw events produced by the Exchange or the Real Time Market Data, but also for complex events generated by trading engines that are input for other engines at a higher level of the hierarchy.
• *real time order:* the information produced by the Exchange or the Real Time Market Data must be delivered in a real time order to all Trading Engine Systems. Such a requirement is fundamental to ensure a fair electronic market service to all clients. As an example, consider the three events: e_1 = *"the price of IBM is 0.5% higher than its average price"*, e_2 = *"the price of Apple is 0.8% lower than the price of IBM"* and e_3 = *"buy 10000 Microsoft's shares if event e_1 is happened before event e_2"*. Now let consider the Trading Engine Systems of two different firms A and B, both interested in e_1, e_2 and e_3. If one of the firms, say A, delivers events out-of-real time order, for example e_2 before e_1, then the event e_3 is not verified and the algorithm does not buy Microsoft's shares. This clearly leads to an unfair market between the firms A and B, that, on the contrary, delivered e_1 and e_2 in the correct real time order.

1.2.3 Active Database in Cloud Computing

In the last few years, cloud computing emerged as a technology to provide resources on-demand and as a service over the Internet. Users can access these resources anytime and anywhere, both from desktops or mobile platforms. Amazon EC2, Google AppEngine, Microsoft's Azure are just a few examples of cloud architectures that provide services ranging from storage and application development to high speed computing platform. A public cloud is typically a complex infrastructure composed by one or more data centers, where a huge number of services runs on a large amount of hardware. A meaningful example of complex infrastructure is represented by the eBay architecture: in order to

provide scalability, manageability and cost reduction, eBay made a functional segmentation of its enterprise into multiple disjoint subsystems: Users, Item, Transaction, Product, Account, Feedback (vertical division). Each subsystem is further divided into chunks (horizontal division) to parallelize the handling of requests within these [99].

The fundamental problem that arises from this segmentation is the maintenance of data consistency among all the chunks in which a database is partitioned. Figure 1.4 depicts a cloud architecture where users, data management applications and the chunks themselves send events to consistently update all replicas of the same database.

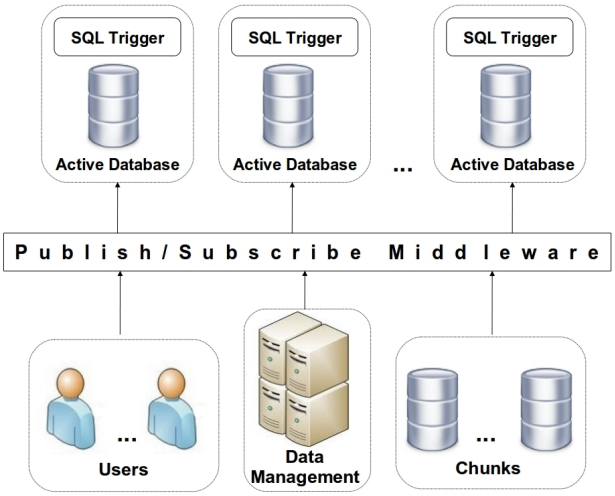


Figure 1.4: Cloud architecture with several sources that send updates to a set of replicated active databases. SQL Triggers execute persistent queries for security monitoring, alerting, statistics gathering and authorization.

Replicas (database chunks) can be viewed as active databases [111] that store all the same information. On top of them, SQL Triggers execute persistent queries to audit changes, enforce and execute business rules, replicate data, enhance performance, monitor the application and gather statistics. Inconsistency could lead to detect some pattern in a chunk hosted in a data center, while leaving the pattern undetected in another data center. To prevent inconsistency, a typical approach is to use transactional ACID-based mechanism. However, this introduces an unsubstainable load of interactions and sycnhronizations (i.e., locks) among cloud nodes that may also hamper the scalability of the system [29]. This is why major cloud providers are moving towards a decentralized convergence behavior in which cloud nodes are

maintained in transiently divergent states, from which they will converge to a consistent state over time. This behavior is known as *eventual consistency* [109]: after an update completes, the system does not guarantee that subsequent accesses will return the updated value; there is an *inconsistency window* that represents the time period between an update and the moment in which any observer will always see the updated value.

A simple way to implement an eventual consistency algorithm is to use a best effort data dissemination service and rollback techniques that allow replicas to *correct* a wrong order. However, even if this solution avoids a strong coordination among nodes that would increase the risk that the whole cloud infrastructure may begin to thrash [29], it ensures that all replicas will be consistent only after a time t. Before t the result of persistent queries is unpredictable: in fact, the same pattern may be detected just by a subset of SQL Triggers, due to the inconsistency of data stored in the databases.

To prevent this problem, the publish/subscribe middleware has to convey the high volume of events generated by sources in a reliable and totally ordered fashion. This two properties are fundamental for ensuring consistency without locking: reliability guarantees that all replicas will receive the same set of messages, while total order ensures that messages will be delivered in the same order by all receivers. This avoids, or at least reduces, the need for rollback techniques, because all chunks will see the same ordered sequence of updates, also preventing the unpredictability in persistency query results.

Reliable delivery and total ordering can be obtained at expenses of a timely delivery. In fact, as observed in [29], it does not matter how fast the protocol is; the aim is to avoid self-synchronization mechanisms among cloud nodes.

Required middleware properties. The publish/subscribe middleware used to support data consistency in cloud computing should guarantee the following properties:

• *reliable delivery:* each operation on a database triggers an update event that must be notified to all interest database chunks. The lack of reliability can have a strong impact on the final result of an operation. As an example, let us consider an auction bid with two bidders that issue an offer for the same product. The two bids represent events that must be notified to a set of database replicas. If one of the two events gets lost, then the outcome of the bid may not consider that offer, preventing the user from a possible victory.

• *total order:* in order to keep coherent copies of data in database chunks, events should be delivered in total ordering as defined in [45]. It does not matter that event deliveries follow the real time ordering of their emission, what matters is that every pair of events delivered by a pair of database chunks are delivered in the same order.

1.2.4 Flight plans exchange in the Air Traffic Control domain

To date, Air Traffic Control (ATC) has been a service provided by ground-based controllers to direct aircrafts *en route* and on the ground. The European airspace has been divided into several volumes, and the duty of controlling flights in a given volume has been assigned to local Area Control Centers (ACCs). *En route* management, departing and landing operations, surveillance and flight data collection has been performed in isolation by local ACCs only when an aircraft was close to an airport. While the traditional ATM framework does not allow the cooperation among ACCs, the future SESAR framework, introduced in Section 1.1, will be architected by implementing a seamless control of aircrafts over multiple interconnected ACCs. As such, the primary purposes of the new ATC system will be:

- to manage a higher volume of avionic traffic;
- to separate aircrafts to prevent collisions;
- to organize and expedite the traffic flow;
- to provide information and other support for pilots;
- to provide advice information for a safe ad efficient conduct of flights;
- to alert search and rescue bodies when necessary.

The SESAR architecture is depicted in Figure 1.5.

The core is represented by SWIM, a middleware solution that federates into a SoS legacy and geographically sparse ATM systems, each one implementing domain specific functionalities. SWIM realizes the many-to-many information distribution among geographically dispersed sources, which collaboratively update the same piece of information, and many destinations, which need to maintain situational awareness with respect to information changes. Air traffic controllers, airports, weather stations, pilots and radars are the main entities in the ATC domain, each of them being both producer and consumer of data. The information spread by the SWIM platform contains flight plans, i.e., a series of information such as departure and arrival airport, the estimated time of arrival, the route carried by the aircraft, possible alternative airports in case of bed weather, etc.

Required middleware properties. With respect to the properties defined in Section 1.1, the publish/subscribe middleware for the ATC scenario has to satisfy:
• *reliable delivery:* each event conveyed by the middleware has to be notified to all interested subscribers. As an example, consider flight plans exchanged

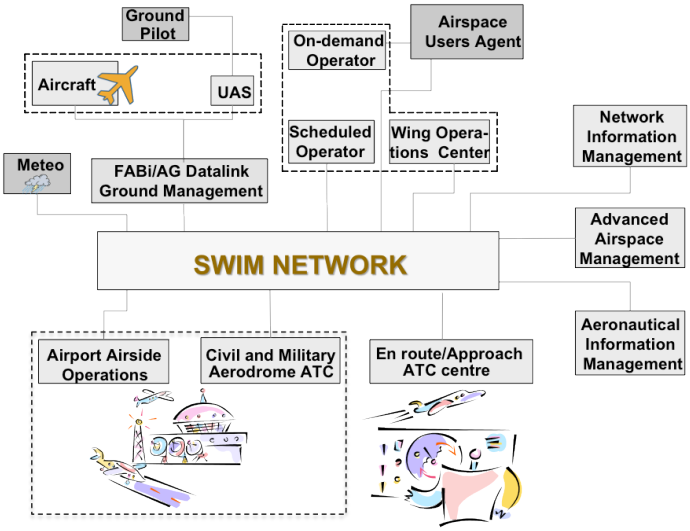


Figure 1.5: The SESAR architecture as a federation of systems interconnected by the SWIM network.

between several flight processors. This data contains information about the trajectory and the coordinates of an aircraft *en route*; if such a message is lost, a flight processor may not be able to infer the current position of an aircraft and to detect possible collisions.

• *timely delivery:* the information must be prepared in real time, meaning that the right information is delivered to the right place at the right time during the entire period of the flight. The service must ensure high performance, and it must therefore be capable of delivering very high volumes of data with very low latency. An event delivered too late can have the same consequence of a loss.

• *causal order:* the publish/subscribe middleware has to respect the causality relationship among events. The violation of such a property, in fact, may lead to several anomalies, as the one described in the example of Figure 1.6.

Consider two ATCs, one in France and the other one in The Netherlands, and an aircraft currently on the Dutch airspace, tracked by the ATC in The Netherlands. The ATC in France publishes the event e_1 = *"The Paris airport is closed due to heavy snow"*, while, after the delivery of e_1, the aircraft publishes the event e_2 = *"Emergency landing request"*. Because the ATC in The Netherlands did not deliver event e_1 before e_2, it is not aware about the closure of the Paris airport, and publishes the event e_3 = *"Emergency landing at Paris airport"*. This obviously generates inconsistent information to the

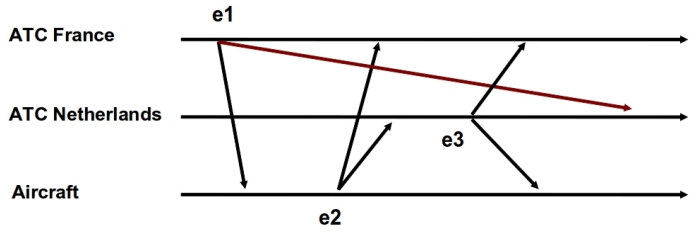


Figure 1.6: A violation of the causal order among events can lead to inconsistencies: the delay in the delivery of message e_1 prevents the ATC in The Netherlands to have a correct picture of the situation.

aircraft, leading it to a possible wrong decision.

1.3 Requirements analysis

In Table 1.1 we summarize the requirements of the analyzed case studies with respect to reliability, timeliness and ordering policies.

↓ **QoS policies**		Coll. sec.	Stock market	Active DB	ATC
Delivery	Best effort				
	Reliable	✓	✓	✓	✓
Ordering	FIFO	✓			
	Causal				✓
	Total			✓	
	Real time		✓		
Timeliness		*soft*	*hard*		*hard*

Table 1.1: Quality of Service requirements for the analyzed case studies.

Looking closely at Table 1.1, we notice that three out of the four case studies we have analyzed require the timeliness property. However, the electronic market and the air traffic control are *mission-critical* applications, i.e., they impose a very tight latency bound, typically of the order of milliseconds (referred to as *hard* in Table 1.1). Collaborative security, instead, imposes a less stringent time bound, typically of the order of several seconds (referred

to as *soft* in Table 1.1). However, timeliness does not really matters for active database in cloud computing. In this scenario, timeliness is traded for scalability [29]: as such, consistency has to be ensured by avoiding the usage of locking algorithms that would impose an unsustainable load on the system due to the synchronization of cloud nodes. To this end, total ordering is of fundamental importance to guarantee consistency in all database replicas, preventing active queries from unpredictable results. Ordering is an important requirement also for collaborative security, stock market and ATC. However, all of them need different ordering policies. The stock market requires a real time order to detect temporal relationships among events occurred at different sources. The collaborative security scenario, instead, does not correlate events coming from different source: collaborative processing engines are used to detect possible ongoing port scanning attacks on single machines. As such, just a per-source FIFO ordering is required. Finally, causal order is required in the ATC domain, to maintain causality relationship among events.

Also reliability matters for all of the described case studies, as a missing information could have a disruptive impact on the detection of an ongoing attack, on the stock trading, on consistent updates and on the analysis of aircrafts trajectory. However, it is worth mentioning that several applications in the ATC domain, such as weather conditions monitoring or radar tracking, can require just a best effort delivery. Indeed, due to the periodic updates sent by sources, the loss of an event has no impact on the computation (a new sample simply overwrites the old value).

1.4 QoS in publish/subscribe

In this Section, we analyze the QoS policies guaranteed by current publish/subscribe middleware implementations and compare them with the results reported in Table 1.1, in order to infer if they are able to satisfy, and to what extent, the requirements of the applications. This analysis can be considered orthogonal to previous surveys on publish/subscribe systems [21, 52, 80]. The goal of [52], in fact, is to introduce the paradigm and to use the time, space and synchronization decoupling properties of the publish/subscribe to compare it with traditional interaction paradigms. Authors in [21], instead, introduce a general architectural model that decomposes a publish/subscribe system into three layers: network protocols, overlay infrastructure and event routing. Then, the paper surveys current existing publish/subscribe middleware for wired and mobile networks and positions them within the proposed architecture. Finally, the work in [80] proposes a survey of publish/subscribe systems considering overlay topology, matching algorithms and other aspects such as reliability (by means of TCP links and path redundancy) and security.

A publish/subscribe middleware can sport two different event selection models: channel-based or content-based. The channel-based model, together with its close sibling, the topic-based model, assumes that every event injected in the system is completely characterized by a name, representing the channel or topic it is published in; processes interested in receiving events can declare their interest in just a subset of all the published information by joining the channels where these events are expected to be injected. Examples of channel- or topic-based middleware are: Data Distribution Service (DDS) [3], Java Message Service (JMS) [104], TIBCO Rendezvous [106], Oracle Stream Advanced Queuing (AQ) [89], SCRIBE [35], IndiQoS [33].

The content-based model is the most general as it assumes that all events are characterized by a set of attributes (with their corresponding values); potential receivers can issue complex subscriptions by applying constraints on available attributes such that the publish/subscribe middleware will deliver them only events satisfying their requirements. Examples of content-based middleware are: JEDI [42], SIENA [34], Gryphon [101], Rebeca [54], Medym [32].

A particular mention for Hermes [92] and PADRES [53]. The former supports two different functionalities: (i) type-based routing, comparable to a topic-based service with the addition of inheritance relationships between event types; (ii) type- and attribute-based routing, that extends the type-based routing with content-based filters on event attributes. The latter is a content-based publish/subscribe middleware supporting workflow management systems, characterized by: (i) a high subscription expressiveness; (ii) a rule engine in each broker that performs subscription-publication matching operations to determine the next hop destination of a message; and (iii) the possibility for subscribers to subscribe future as well as past events.

1.4.1 Publish/subscribe prototypes and commercial systems

Research prototypes. Although many real world applications require support for QoS, as described in Section 1.2, the majority of current research prototypes mainly operates on a best-effort basis [88]. Authors in [83] present a taxonomy of QoS-aware publish/subscribe services with respect to several aspects: timeliness, bandwidth, reliability, delivery semantics and message ordering. Starting from these results, in Table 1.2 we further categorize the QoS policies satisfied by different publish/subscribe solutions based on the properties defined in Section 1.1.

The analysis conducted in [83] stated that Hermes and Medym show reliability properties. However, both solutions guarantee reliability just from

↓ **QoS policies**		JEDI	PADRES	Si-R-SC-H-M	IndiQoS
Delivery	Best effort			✓	✓
	Reliable	✓	✓		
Ordering	FIFO				
	Causal	✓			
	Total				
	Real time				
Timeliness					✓

Table 1.2: Quality of Service policies satisfied by different publish/subscribe middleware. The acronim "Si-R-SC-H-M" refers to Siena, Rebeca, SCRIBE, Hermes and Medym respectively.

the fault-tolerance point of view. In fact, they use techniques to enable event brokers to recover after a failure, or to route information only toward brokers known to be online, but they do not provide support for client-specific delivery requirements as a service guarantee. On the contrary, PADRES additionally tolerates message losses and guarantees publication delivery: overlay path redundancy is exploited both for the routing process and fast information recovery. On the other hand, SCRIBE is a best-effort middleware, that provides extensions to allow specific multicast group to use the reliable TCP transport protocol. Anyway, broker failures are not handled: as such, if a broker fails, a reliable delivery is no more guaranteed, even in presence of a reliable communication protocol. TCP is also used by JEDI. However, it is well-known that the use of TCP in multicast tree overlays exhibits low throughput [22, 29] in practical applications. In addition, the establishment of connections has an inherent mismatch with the decoupling properties of publish/subscribe systems [33].

JEDI is the only publish/subscribe middleware that satisfies an ordered delivery of events, specifically it ensures causal order. It is obtained by means of a *return value*, i.e., a response message that a receiver uses to notify an event delivery to the producer of that event. This mechanism is clearly not scalable in presence of a high number of nodes [42] and a high event rate.

IndiQoS focuses on respecting timeliness in event dissemination. Subscribers specify the latency constraint as an attribute of their subscriptions; brokers take care to reserve a path from subscribers to publishers based on the aggregate traffic they manage, by means of a constant number of deter-

ministic attemps in order to find routes that satisfy those requirements. From the analysis in [83], we can state that IndiQoS also supports bandwidth constraints, while Medym guarantees an exactly-once event delivery. However, bandwidth and delivery semantics are not considered in this thesis. Finally, publish/subscribe middleware such as SIENA and Rebeca are pure best-effort services: differently from SCRIBE, they do not guarantee a reliable delivery even in absence of failures (i.e., they neither provide a fault-tolerant mechanism for brokers nor use a reliable communication protocol).

An analysis similar to the one presented in [83] has also been conducted in [24], where the authors discuss relevant QoS policies and their application to the publish/subscribe model. In particular, the paper focuses on the relationship between QoS and the event notification topology. Authors indentifies two different topologies for the global overlay that connects brokers: *internal overlays*, i.e., dissemination trees within the global overlay, and *external overlays*, i.e., a broadcast overlay for each group. Authors claim that internal overlays present scalability issues due to the presence of rendezvous nodes ([33, 35, 92]): the total number of subscriptions of a node's neighbors is likely to be higher near rendezvous nodes than elsewhere in the network. This increases the load of nodes closer to the source of a multicast tree, and, hence, the overall latency. On the contrary, external overlays, such as in [54], reduce the number of routing hops, and, then, the latency. Different overlays can also build different topologies, to optimize with respect to their group size and requirements. In addition, external overlays ensure message redundancy, because all nodes in a group are interested in all messages, and prevent subscribers of different groups to receive wrong notifications. However, external overlays are not scalable over a high number of groups (brokers have to maintain an overlay for each group they belong to).

Commercial systems. Several commercial systems are nowadays available for implementing the publish/subscribe paradigm in real world applications. Among these systems, we mention TIBCO Rendezvous [106], DDS [3], JMS [104], Oracle Stream Advanced Queuing (AQ) [89] and CORBA Notification Service [1]. Both JMS and AQ can be used for point-to-point (or point-to-multipoint) messaging and for publish/subscribe event dissemination. All these systems differ among them for the QoS policies they offer, as reported in Table 1.3.

DDS allows the control of a rich set of QoS parameters, such as latency budget, reliable delivery, source-based or destination-based event order, event persistency, subscription durability, etc [39]. This makes DDS appealing for several mission critical scenarios, such as air traffic control and military domains, where a piece of information has always to be delivered at the right

↓ **QoS policies**		DDS	JMS	TIBCO	AQ	CORBA
Delivery	Best effort					
	Reliable	✓	✓	✓	✓	✓
Ordering	FIFO	✓		✓		
	Causal					
	Total					
	Real time					
Timeliness		✓		✓		

Table 1.3: Quality of Service policies satisfied by different commercial systems.

place and the right time. However, the configurability of DDS comes at expenses of scalability: DDS works for a small/medium local network, while performance strongly degrades in WAN and over a high number of nodes.

A similar mention for TIBCO Rendezvous, that can be considered the major player in the context of stock market services. Although it does not allow the same QoS control as DDS, TIBCO Rendezvous guarantees reliable delivery by means of retransmissions, FIFO ordering and timely delivery, due to the use of IP multicast. However, even in this case, performance comes at the cost of a difficult deployment in WAN, due to the lack of IP multicast support of legacy network devices.

The only QoS policy ensured by JMS, instead, is reliability, by means of message persistency and subscription durability [39]. The lack of timeliness support, makes it not suitable for mission critical applications; on the contrary, it has been used to implement the collaborative security service described in Section 1.2.1, in the context of the CoMiFin European project [38]. As JMS, also the CORBA Notification Service ensures only reliable delivery. A user can choose between: (i) *transient* (best-effort) subscription, i.e., if a single attempt to deliver an event fails, than the event is lost; (ii) persistent (reliable) subscription, i.e., an event is guaranteed to be eventually delivered. It is worth mentioning that persistent subscriptions do not take into account a strict delivery semantics, i.e., an event can be delivered more than once.

Finally, AQ is a database-integrated service that includes query support, triggers, indexing, transactions, consistency constraints, access control, data management, etc. Due to its tight database coupling, it is suitable to implement active database applications. However, as JMS, AQ provides support just for reliable delivery, by means of persistent messaging.

1.4.2 Application requirements and QoS policies in pub/sub: gap analysis

The QoS requirements addressed by commercial systems, such as DDS and TIBCO Rendezvous, come at expenses of a difficult deployment and/or performance degradation in WAN environment. Hence, these solutions can only be used within local systems, while they do not result well suited to federate local systems in a SoS. In addition, some functionality needs to be added to fully meet all QoS requirements. As an example, DDS and TIBCO Rendezvous provide only FIFO ordering; thus, additional algorithms and event reordering techniques have to be deployed at application level to guarantee causal or real time ordering, as required by ATC or stock market applications.

On the contrary, research prototypes such as SIENA, SCRIBE, Hermes, IndiQoS, are designed to be scalable over large scale systems: the event notification service is typically an application level multicast tree that conveys the information from a publisher to all intended subscribers, without relying on the IP multicast technology. However, we have seen in Table 1.2 that these middleware services do not provide the necessary native support to QoS parameters, in order to federate independent systems in SoS.

From the assessment of the study reported in tables 1.1, 1.2 and 1.3, we can conclude that none of the current research prototypes as well as commercial systems can actually be used for the novel generation of LCCIs without extending its functionalities at application level. As such, in the next Chapters of the thesis we describe a framework that can be posed on top of a publish/subscribe middleware to address ordering, timeliness and reliability for the implementation of applications in large scale systems. In this way, QoS issues do not need to be addressed at application level, so as to leave applications to implement just their native functionalities.

1.5 Challenges

Addressing QoS requirements in SoS over WAN poses unprecedented challenges in the design of algorithms. In this Section we analyze the main challenges that arise when devising protocols for ensuring ordering, timeliness and reliability in large distributed systems.

Ordering. Ordering in single source applications can be trivially achieved by timestamping events with a sequence number. By looking at this number, a receiver can infer if the event can be delivered or delayed. On the contrary, in presence of multiple sources, as in publish/subscribe systems, ensuring or-

dering is more challenging. First of all, several order semantics can be defined: FIFO, causal, total, real time, as done in Section 1.1. Then, while FIFO order can be satisfied by timestamping events as in the case of a single source application, causal and total order can be easily achieved only by relying on a central entity that timestamps events with sequence numbers. The presence of a centralized point clearly poses scalability issues in large scale systems. However, a distributed approach in WAN may be affected by the variation of the transmission delay [79]. This problem in publish/subscribe systems is further exacerbated by the intersection of users' subscriptions, with the risk to deliver the same events in a different order to several subscribers. Finally, real time order requires synchronized clocks: however, the WAN asynchrony and phases of disconnection may hamper the accuracy of the synchronization [79], and, in turn, the correct ordering of events.

Timeliness. Timeliness is a key requirement for mission critical and high throughput applications, that try to maximize the number of tasks that complete within a given deadline. However, the timeliness of a system is typically affected by two factors: (i) the asynchrony of the network, that does not allow to establish an upper bound on the transmission delay, and (ii) the network and nodes failures, that can require retransmission in case of message losses. In particular, the last point evidences a clear trade-off between timely and reliable event notification. To date, ensuring reliability in publish/subscribe middleware without affecting temporal constraints is still an open problem.

Reliability. A typical way to ensure reliability in event dissemination in WAN is to impose a strong or weak feedback loop between sources and destinations. However, the feedback imposes synchronization among nodes, which can create instability in high throughput applications [44]. Examples of system oscillations have been reported in Air Traffic Control systems, Amazon system platform and IP multicast [28]. Decoupling the system nodes, by using asynchronous communications whenever is conceivable and reducing synchronization points as few as possible, are key requirements to effectively support high load and scalable applications [109]. To this end, gossiping techniques can be used to retrofit reliability in event dissemination over WAN. However, due to the probabilistic nature of these algorithms, it is of fundamental importance to properly set the gossip parameters (i.e., number of rounds, number of gossip partners) in order to increase the probability to notify an event to all interested subscribers.

1.6 Contribution

The contribution of the thesis is described in Chapters 3 and 5, where we present novel protocols to address ordering, timeliness and reliability issues. In addition, Chapter 4 shows how the ordering algorithm can be applied to a database cloud to ensure eventual consistency among all database replicas. After Chapter 2, that introduces the system model and node architecture assumed in the remainder of the thesis, we present the main contribution as follows:

Ordering. In Chapter 3 we propose a novel solution for out-of-order notification detection on top of an existing topic-based publish/subscribe middleware. Our solution guarantees that events published on different topics will be either delivered in the same order to all the subscribers of those topics or tagged as out-of-order. The proposed algorithm is completely distributed and is able to scale with the system size while imposing a reasonable cost in terms of notification latency. It is based on a sequencing network of *topic managers* that assign a sequence number to each event published on the topics they manage. In addition, we define a total order relation among topics that determines a one-way sequence of topic managers that collaboratively generate a timestamp for each published event. The subscribers that receive an event, can infer the correct order by looking at the timestamp associated to that event. Our algorithm improves the current state of the art solutions by dynamically handling subscriptions/unsubscriptions and by automatically adapting with respect to topic popularity changes. Indeed, we also propose a mechanism that modifies the total order relation among topics at run-time, in order to reduce the timestamp size and to minimize the latency overhead imposed by the ordering algorithm. Finally, we conducted a simulation-based experimental analysis in a large scale system to evaluate the performance of our solution, and a prototype-based study in a small scale WAN network that compares the proposed algorithm to a solution based on the JGroups [67] communication toolkit.

Ordering application to a database cloud. In Chapter 4 we apply the total order algorithm described in Chapter 3 to a database cloud to ensure eventual consistency among database replicas. We compare the performance of our protocol in terms of notification latency and percentage of rollback operations caused by out-of-order event delivery to a similar solution based on a gossip algorithm for data dissemination [16]. A prototype-based experimental study conducted on a cluster of real nodes shows that the cooperative procedure used by our algorithm to generate a timestamp is fundamental to decrease the number of out-of-order notifications, and, then, the percentage of

rollback operations. However, this results, is obtained at the cost of a higher latency when the number of topics increases.

Timeliness and reliability. In Chapter 5 we describe a solution that provides both reliability and timeliness in publish/subscribe systems. Although these two requirements can be separately addressed, we decided to present a joint solution because there exists a rich literature on algorithms for reliable communications, but a solution for providing fault-tolerance without violating the timeliness requirements still lacks. Indeed, fault tolerance is typically achieved at the cost of severe performance fluctuations, or timeliness is always obtained by softening the fault-tolerance requirements. On the contrary, we propose to fulfill this lack by combining two different approaches, namely random linear coding and gossiping, able to satisfy timeliness and reliability requirements, respectively. We consider the scenario in which a publisher sends information to a set of subscribers, which in turn apply a gossiping strategy to recover from possible lost events. We also provide a theoretical model to evaluate the potential benefit of coding on the information delivery performance. These results are confirmed by an experimental analysis conducted on a real air traffic control workload, which evidences how coding mitigates latency and overhead penalties to ensure reliable event notification.

The contributions listed above and basic ideas from which this work has started are partially contained in the following papers: [12, 13, 50, 51]. In addition, during the Ph.D., the author also worked in the field of peer-to-peer systems [18, 19] and next generation networks [9, 94].

Chapter 2

System model and node architecture

In this Chapter, we describe the system model and the node architecture that we assume in the remainder of the thesis. Some integrations will be presented and discussed in Chapters 3 and 5.

2.1 System model

We consider a system populated by geographically sparse and loosely synchronized nodes deployed across different administrative domains; they are connected by means of an overlay network built on top of a wide area network, for example the Internet. System nodes can access a coarse grain common clock such as Network Time Protocol (NTP) and might have knowledge about their geographical location. This means that information can be labeled with timing and geographical timestamps. Note that due to the typical unpredictable delays of loosely coupled distributed systems like the Internet, this kind of synchronization cannot be used to reliably and totally order events produced by independent sources [79]. The number of nodes in the system depends on the applications: for example, in the air traffic control scenario we can have 3/4 different systems per nation federated in a SoS, where nations are those of the European Community (currently 27); Thus, the scale of the overall SoS can vary from few dozens to several thousand depending on the specific context. Figure 2.1 illustrates a possible scenario of interest: it represents three US airports, New York, Chicago and Las Vegas, each of which is an independent air

traffic control system. A SoS deployed over the US landscape federates these three systems by connecting some of their nodes by means of a middleware service, that allows flight plans information sharing (dashed lines).

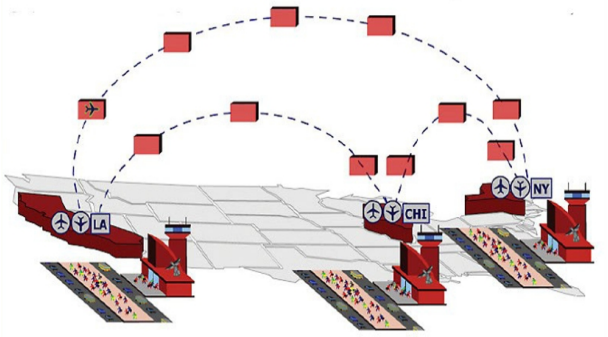


Figure 2.1: An example of a SoS composed by three US airports.

2.2 Node architecture

Each node in the system implements the architecture depicted in Figure 5.1. It is composed by three building blocks: application, QoS layer and Event Notification Service (ENS).

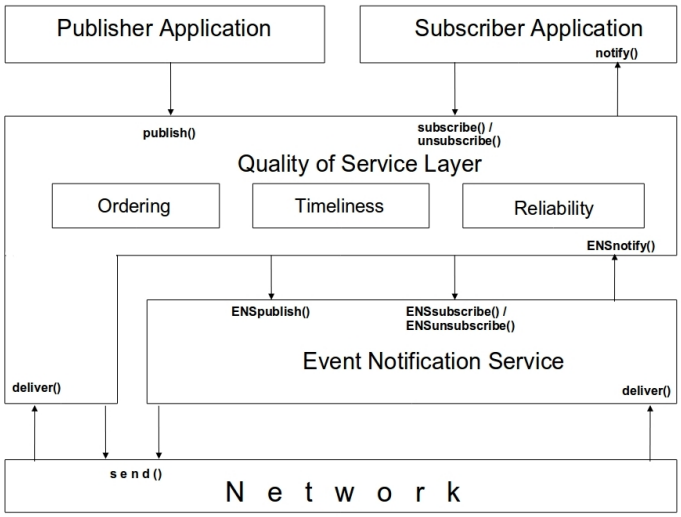


Figure 2.2: Architecture implemented by all SoS nodes.

Application. The architecture we designed is targeted for several applications deployed over a large scale network, for example maritime surveillance, air traffic control, collaborative security, next generation intelligence platform, stock market, etc. We individuate two distinct roles: information producers, i.e. *publishers*, and information consumers, i.e. *subscribers*.

Producers of information can be: sensors that capture data from an environment (temperature, humidity, enlightment); systems or devices, for example firewalls, that produce log files for monitoring applications; radars or stock market sites that generate periodic updates; data sources that update database in a cloud.

On the contrary, consumers of information are: systems that infer environmental conditions (fires, floodings) by analyzing data detected by sensors; trading applications that buy and sell actions based on data produced by stock market sites; flight processors that analyze information about flight plans.

Finally, several applications can be both publishers and subscribers. As an example, the processing engines introduced in Section 1.2.1 can be subscribers of raw events generated by firewalls of Financial Infrastructures, and at the same time, they can also be publishers of complex events obtained by correlating raw events coming from different sources.

QoS layer. As stated in Section 1.1, the publish/subscribe paradigm can be really the winning solution for event dissemination in SoS only if the middleware is able to offer a means to provide QoS. Due to the best effort nature of publish/subscriber middleware, also discussed in Section 1.4, we wrap the ENS with a *Quality of Service Layer* that ensures some of the QoS properties indroduced in Section 1.1, specifically total ordering, timeliness and reliability. We extensively described in Sections 1.2, 1.4 and 1.5 how these aspects affect the implementation of large scale applications.

Blocks in the QoS layer are independent and can be combined together in order to satisfy the QoS requirements of the specific application. The QoS layer intercepts events published by the application or notified by the ENS, and runs one or more algorithms depending on the selected QoS blocks. In Chapter 5 we will show how two independent blocks, namely timeliness and reliability, can be used together without further modifications. The QoS layer may also require additional interaction among nodes, and it can be done by accessing a point-to-point communication primitive that can be offered by the operating system or by other solution like an overlay network. Finally, when the QoS layer has completed its task, it publishes the events on the ENS or notifies them to the subscriber application.

Event Notification Service. The communication model used in our architecture follows the event-based publish/subscribe paradigm. The interaction among publishers and subscribers is mediated by a distributed topic-based *Event Notification Service (ENS)*, that provides the standard interface of a publish/subscribe middleware [91]:

- *publish()*: invoked by publishers to publish events in the system;
- *subscribe()*: invoked by subscribers to declare an interest in a topic of events;
- *unsubscribe()*: invoked by subscribers to unsubscribe from a previously subscribed topic;
- *notify()*: invoked by the ENS to deliver events to subscribers according to their topics of interest.

The intrisic decoupling properties of the publish/subscribe paradigm make it appealing for satisfying scalability requirements of large scale application as previously mentioned. In addition, it is worth noticing that the presented QoS layer exposes the same interface of the ENS; thus neither the applications, nor the ENS must be changed in order to work with our framework. This aspect, joint with the facts that the QoS layer does not require any degree of synchronization among participants and does not violate the anonymity principle of the publish/subscribe paradigm, makes our framework independent from the underlying ENS implementation, such that all the current solutions described in Section 1.4 can be employed in practice.

Chapter 3

Ordering issues

A distributed ENS can route events toward subscribers using multiple paths with different lengths and latencies; as a consequence, subscribers can receive events out of order. However, many research efforts in publish/subscribe systems focused on reliability and performance aspects with few contributions in the area of event ordering [58, 79, 82, 84]. Defining a coherent specification for notification ordering is a fundamental step for the case studies analyzed in Section 1.2 and for a wide range of other applications such as online games [26], messaging, or those based on composite event detection [93]. All these applications assign a semantics to the order in which events are notified; therefore, it is important that a notification ordering is specified and that the underlying ENS is able to guarantee its adherence to this specification or, at least, to provide hints about which subsets of notifications are guaranteed to be notified "in order". As an example, in composite event detection applications, ordering is fundamental to recognize pattern of sequences on the input event stream [93].

In this Chapter, we address the following simple ordering problem: how to guarantee that two subscribers sharing (at least) two same subscriptions are notified about events matching those subscriptions in the same order. While the above ordering problem stems from the simple rationale that two participants should always see the notification of two events in the same order, its enforcement in distributed ENSs is far from being trivial. Violations to the ordering property can easily arise due to the fact that two events, possibly published by different publishers, can follow distinct paths through the ENS before reaching the point where they will be notified to the final recipients[1].

[1] Non-determinism in the form of unpredictable network latencies and message losses can easily exacerbate the problem.

The impact of this problem can be easily seen by running a simple experiment executed on a toy application where two subscribers receive events published by two sources on two different topics, and check the occurrence of a specific notification pattern (e.g. the sequence of notifications $e \rightarrow e' \rightarrow e''$); the results we obtained by running this test in a simple setting where events are diffused using SCRIBE show how the ENS notifies events in a best-effort fashion without providing any form of ordering, thus allowing the two subscribers to coherently detect only about 35% of the patterns (further details on this test are reported in Section 3.5). Current approaches to solve this problem either (i) use hardware-based synchronization solutions to timestamp events [79] or (ii) implement total order [58] among all the receiving participants by trading performance for strong ordering guarantees, or (iii) give up some ordering aspects only guaranteeing per-source ordering [3] or, finally, (iv) require complex offline set-ups that must be continuously updated when subscribers change their interests [82].

In this Chapter we present a novel algorithm for out-of-order notification detection in distributed topic-based systems. Our solutions, encapsulated within a software component that can be deployed on top of any existing topic-based ENS, transparently delivers events notified by the ENS to the application layer and is able to deterministically tag every event whose notification violates the following *total notification order* property: if two independent subscribers are notified about the same two events, then these two events will be notified to them in the same order[2]. Out-of-order detection is realized by comparing logical timestamps that our algorithm automatically generates and attaches to events. The algorithm can use a configurable buffer to re-order events prior to notification as this can easily reduce the number out-of-order notifications. The algorithm performance has been analyzed through an extensive experimental study whose results show how our solution has a small impact on the event diffusion latency and the ability to dynamically adapt its behaviour to the current topic popularity distribution. Finally, we developed a prototype implementation of our algorithm, whose performance has been compared to a solution based on the JGroups [67] toolkit.

3.1 System model and problem statement

We assume that publishers and subscribers share a common knowledge about a set of available topics, and that each subscriber issues a subscription containing the set of topics it is interested in. An event e published on a topic T matches

[2]Note that this delivery order does not necessarily correspond to the real-time event production order.

a subscription S if and only if $T \in S$; if this happens, the corresponding subscriber must be notified about e.

In addition, in order to simplify the description of our solution, we will initially assume that our system works on top of a reliable communication substrate, that all communication links deliver messages in FIFO order, and that all processes are correct. Some of these assumptions will be removed or relaxed in Section 3.4.

The ordering property we want to enforce is defined as follows:

Property 1. TOTAL NOTIFICATION ORDER (TNO). *Let e_i and e_j be two distinct events notified to a subscriber s. If e_i is notified to s before e_j, no subscriber will be notified about e_i after being notified about e_j.*

Note that this definition matches the definition of *Weak Total Order* given in [14] in the context of total order specifications [45]. Differently from those specifications, we do not consider any form of deterministic agreement (uniform or not uniform) because here we are only interested in designing an ordering layer to be transparently plugged on top of a generic ENS which can provide different reliability and agreement properties.

Guaranteeing TNO in a distributed setting is a non trivial task. Consider, for example, the toy system depicted in Figure 3.1: the six black dots represent processes constituting the ENS, the white dots on the left (p_1 and p_2) are two publishers and those on the right (s_1 and s_2) are two subscribers.

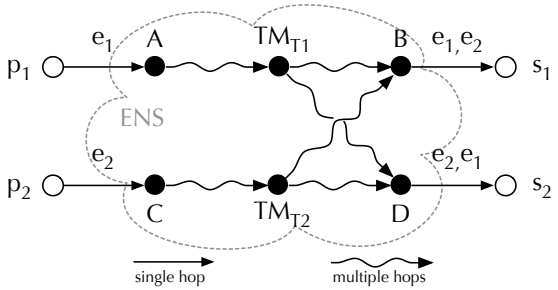


Figure 3.1: An example showing how notifications can be performed out of order in a distributed event notification service.

A simple solution for ordering events published on a specific topic is based on the usage of topic managers: a single node in the ENS is elected as a "sequencer" for all the events published in that topic. In our example TM_{T1}

acts as the topic manager node for topic $T1$, receiving all the events published in $T1$ (i.e., event e_1 published by p_1), adding a sequence number to them, and then routing the events toward the intended destinations (i.e., s_1 and s_2 notified by nodes B and D).

However, this simple approach is not useful when the subscriptions intersect in multiple topics. For example, assume that both s_1 and s_2 are subscribed to $T1$ and $T2$. In this case the sequence numbers attached by TM_{T1} and TM_{T2} would be completely uncorrelated and thus useless to check for a correct notification order on the subscribers' side. The obvious solution, i.e. having a single topic manager for all the topics, has important scalability and reliability drawbacks and cannot thus be considered as a realistic alternative.

3.2 The event ordering algorithm

This Section first describes how the proposed solution can fit within an existing architecture based on publish/subscribe interactions, and then, details the algorithm that implements the solution.

3.2.1 Architectural aspects

Our solution assumes that all participants to the system (publishers and subscribers) are equipped with an *Ordering module* that implements the algorithm described in the next Section (see Figure 3.2). This module mediates the interactions between application level software components, that act as information producers (publisher applications) or consumers (subscriber applications), and a standard ENS.

We also assume that the set of available topics is fixed and a precedence relationship $\rightarrow$ holds among topic identifiers inducing a total order on them[3]. Moreover, we assume that there is a method to univocally map a topic T to a single participating node in the system that will act as the *topic manager* for that topic (TM_T). This latter assumption can be satisfied in several different ways, i.e., through a static mapping provided as a configuration parameter or using a distributed hash table as in rendez-vous based publish/subscribe systems [35]. In the following, whenever there is no ambiguity, we will use the terms *publisher* and *subscriber* to refer the parts of our ordering module located respectively at the publisher and at the subscriber site.

[3]This assumption will be relaxed in Section 3.4 where we will show how this order can be changed at run-time in order to improve the performance.

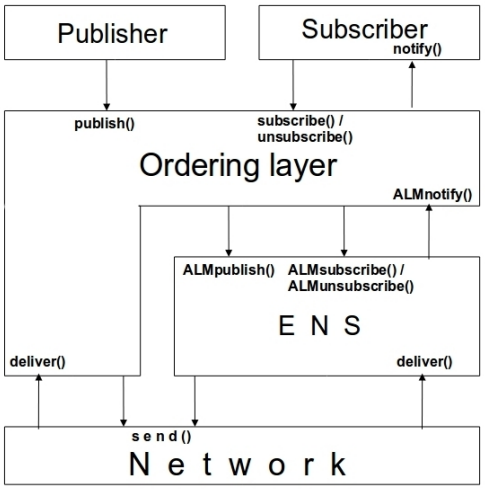


Figure 3.2: Architectural view that shows how the ordering module acts as a mediating software layer between the applications and an existing event notification service.

3.2.2 Algorithm description

The basic idea behind the algorithm is to assign a logical timestamp to each event. By looking at a timestamp, a subscriber must be able to decide if the event can be notified or it needs a tag, witnessing that it was received out of order. The notified application will then take a decision on how out-of-order events must be treated. The algorithm to be executed when an event is published is split in three phases: (i) timestamp generation, where a timestamp is generated for the event, (ii) event diffusion, where the ENS delivers the event and its timestamp to all the intended subscribers, and (iii) event notification, where subscribers, by looking at the timestamp content, decide if the event must be tagged as out-of-order before notifying it (Figure 3.3). The algorithm uses only local information maintained by each process: a topic manager TM_T stores all subscriptions containing topic T and a sequence number that counts the number of events published on T, while each subscriber stores its subscription S and a set containing the sequence number of the last event notified on T, for each topic $T \in S$ (i.e., it maintains a local subscription clock).

The first phase is started by a publisher requiring the creation of a timestamp for a new event e published on a topic T. The timestamp creation for e is carried out by the subset of TMs associated to topics belonging to the *sequencing group* of T (namely $\mathcal{SG}_T$). $\mathcal{SG}_T$ is a set of topics including all T' such that (i) $T' \rightarrow T$ and (ii) there are at least two subscriptions including both T

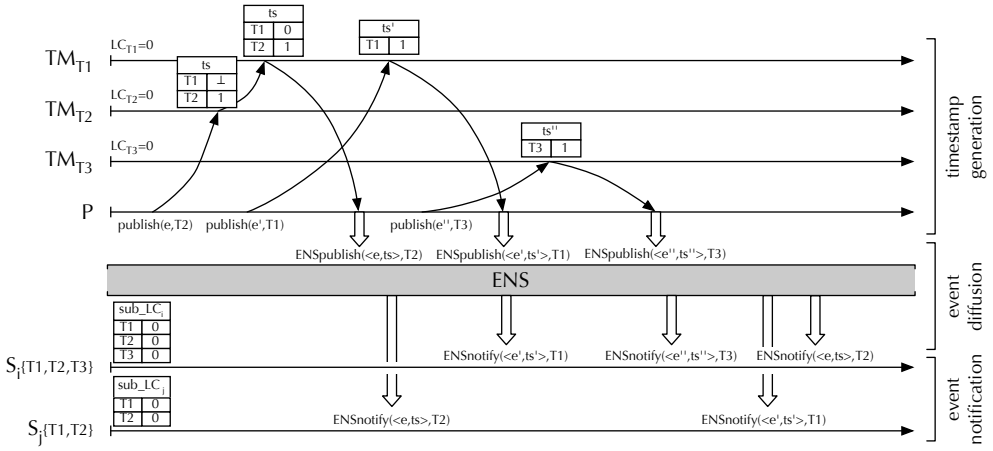


Figure 3.3: Example of a system run with two subscribers, S_i and S_j, and a publisher P.

and T'. The timestamp generation procedure is started by TM_T when it receives the request of a timestamp generation from the publisher. TM_T creates the structure of the timestamp adding one entry for each topic T' such that $T' \in \mathcal{SG}_T$, stores T's current sequence number and forwards the timestamp to the TM associated to the first topic in $\mathcal{SG}_T$ that precedes T according to the precedence relation $\rightarrow$. Note that, given a specific order $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ among topics, the timestamp generation flow proceeds in the opposite direction (i.e., given a topic T_i, TM_{T_i} will fill in the timestamp and forward it to some TM_{T_j} such that $T_j \rightarrow T_i$). The receiving TM adds the sequence number for the topic it manages to the timestamp and sends it to the TM associated to the next topic in $\mathcal{SG}_T$. When the last TM completes the timestamp, it is returned to the publisher that will publish the event on the ENS together with the timestamp, starting the event diffusion phase. This collaboration among several TMs in the creation of a timestamp is fundamental to totally order events published on their corresponding topics, and thus it avoids possible TNO violations like the one shown at the end of Section 3.1. Referring to the example in Figure 3.3, where the topic order is $T_1 \rightarrow T_2 \rightarrow T_3$, the publisher P publishes an event e on topic $T2$ and asks TM_{T2} to create the timestamp. In this case $\mathcal{SG}_{T2} = \{T2, T1\}$. Therefore TM_{T2} creates the structure of the timestamp with entries for topics $T2$ and $T1$, puts its sequence number in the timestamp and forwards it to TM_{T1} that, in turn, will complete the timestamp and return it to P. Finally P publishes both e and its timestamp on

the ENS.

In the event notification phase, once an event e and its timestamp are notified by the ENS, the subscriber checks if the timestamp attached to the event is coherent with the event order maintained through the local subscription clock. If so, the event is notified to the application, otherwise it is tagged to let the application be aware that it is notified out-of-order (cfr. paragraph NOTIFY() Operation in the following). Once an application is notified about an unordered event, it will decide, according to its specific requirements, if the order inversion can be tolerated or if the event must be discarded.

Note that, given a topic T, its sequencing group is defined according to all subscriptions containing T. As a consequence, every new subscription (or unsubscription) to topic T induces a modification of the subset of topic managers involved in the timestamp generation phase. Thus, when a topic T is added to a subscription S, each topic manager $TM_{T'}$ associated to a topic T' in S must be advertised about the change of the subscription to avoid possible TNO violation. To this aim, the subscriber creates an empty subscription timestamp, containing one entry for each topic in the subscription. Similarly to event timestamp, each entry of the subscription timestamp is filled in by the corresponding topic manager. In addition, receiving the request, each topic manager TM_T updates the list of subscriptions it knows. When the sequence number of the last topic manager belonging to the subscription is added, the latter sends the complete timestamp to the subscriber, that, in turn, resets its local subscription clock. The same approach is used to unsubscribe a topic. When a subscriber is no longer interested in events of a topic T, it advertises the change on the subscription to topic managers managing topics in its subscription. Receiving the unsubscription, a topic manager just updates the set of subscriptions it knows.

In the following, before describing the algorithm operations in detail, we first provide the definition of a *timestamp associated to an event e* and then we specify an *order relation between two timestamps.*

Definition 1. *Let e be an event published on a topic T, $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be the topic ordering and let $\mathcal{SG}_T$ be the sequencing group of T. A timestamp ts_e for e is a set of pairs $< T_i, sn_i >$ ordered according to the precedence relation $\rightarrow$, where $T_i \in \mathcal{SG}_T$ is a topic identifier and sn_i is e's sequence number for topic T_i.*

Definition 2. *Let ts_e and $ts_{e'}$ be two timestamps associated with two different*

events e and e'. We say that ts_e and $ts_{e'}$ are comparable *if there exists at least one topic identifier included both in ts_e and $ts_{e'}$ (i.e. $\exists\ t_{id}, i, j \mid (< t_{id}, i > \in ts_e) \wedge (< t_{id}, j > \in ts_{e'})$).*

From the two definitions above, it is easy to see that given two events e and e' published respectively on topic T and T', the corresponding timestamps ts_e and $ts_{e'}$ are comparable if and only if $\mathcal{SG}_T \cap \mathcal{SG}_{T'} \neq \emptyset$.

Definition 3. *Let ts_e and $ts_{e'}$ be two timestamps associated with two different events e and e'. We say that ts_e* is smaller *than $ts_{e'}$ (i.e. $ts_e < ts_{e'}$) if*

1. *ts_e and $ts_{e'}$ are comparable and*
2. *$\forall < t_{id}, sn > \in ts_e \mid \exists < t_{id}, sn' > \in ts_{e'},\ sn \leq sn'$ and*
3. *$\exists < t_{id}, sn > \in ts_e \mid \exists < t_{id}, sn' > \in ts_{e'},\ sn < sn'$*

As an example, in Figure 3.3 we show the timestamps for three published events e, e' and e''. Considering the timestamp ts associated to e and the timestamp ts' associated to e' we have that they are comparable (there is at least one topic, i.e., T_1, belonging to both ts and ts'); on the contrary, ts and ts'' (or even ts' and ts'') are not comparable. Moreover, considering ts and ts', we have that $ts < ts'$.

Local data structures to each publisher p_i: each publisher maintains locally the following data structures:

- id_e: is a unique identifier associated to each event produced by p_i.
- $outgoingEvents_i$: a set variable, initially empty, storing the events indexed by event id that are published by the upper application layer, and that are waiting for being published on the ENS.

Local data structures to each subscriber s_i: each subscriber maintains locally the following data structures:

- $subs_i$: a set variable storing topics subscribed by p_i;
- sub_LC_i: a set of pairs $< T_i, sn_i >$, where T_i is a topic identifier and sn_i is an integer value; sub_LC_i contains a pair for each topic $T_i \in subs_i$. Initially, for each topic $T_i \in subs_i$ the corresponding sequence number is $\perp$.

Local data structures to each topic manager TM_{T_i}: to simplify the notation, we assume that each topic manager TM_i is responsible for one topic T_i[4]. Each topic manager maintains locally the following data structures:

- LC_{T_i}: is an integer value representing the sequence number associated to topic T_i, initially 0.
- $externalSubs_{T_i}$: a set of pairs $< id, sub >$ where sub is a subscription (i.e., a set of topics $\{T_j, T_k \dots T_h\}$) and id is the subscriber identifier. Such a set contains all the subscriptions that include T_i.

As an example, let us consider the system depicted in Figure 3.3. Let $S_i = \{T1, T2, T3\}$ and $S_j = \{T1, T2\}$ be respectively the two subscriptions of s_i and s_j. The three variables $externalSubs$ maintained by each topic manager are respectively: $externalSubs_{T1} = \{< i, S_i >, < j, S_j >\}$, $externalSubs_{T2} = \{< i, S_i >, < j, S_j >\}$ and $externalSubs_{T3} = \{< i, S_i >\}$.

PUBLISH() **Operation.** The algorithm for a PUBLISH() operation is shown in Figure 3.5. To simplify its pseudo-code, we defined the following basic functions:

- generateUniqueEventID(e): generates a locally unique identifier for a specific event e.
- next(ts, T): given a timestamp ts and a topic identifier T, the function returns the identifier of the topic T' preceding T in the timestamp ts, according to the precedence relation $\rightarrow$; if a *null* value is passed as topic identifier, the function returns the last topic identifier contained in the timestamp.
- getTopicRespAddress(T): returns the network address of the topic manager TM_T responsible for topic T.
- update(ts, T, LC_T): updates the event timestamp ts changing the pair $< T, - >$ with the pair $< T, LC_T >$.

In addition, we have defined a more complex function, namely createPubTimestamp $(T, externalSubs_T)$, that generates an empty timestamp for a generic event published on topic T by considering the set of subscriptions containing T (i.e., subscriptions stored in $externalSubs_T$). The pseudo-code of the function is shown in Figure 3.4.

[4]This assumption does not limit the applicability of our solution. If the topic manager is responsible for more than one topic, its local variables must be duplicated, one for each topic.

```
function createPubTimestamp(T, externalSubs):

(01) for each < −, s >∈ externalSubs do SG_T ← SG_T ∪ s; endfor
(02) sort (SG_T);
(03) for each T_j ∈ SG_T : T_i → T_j do SG_T ← SG_T/{T_j}; endfor
(04) for each T_j ≠ T_i ∈ SG_T do
(05)     let S = {s ∈ externalSubs|T_j ∈ s};
(06)     if (|S| ≤ 1) then SG_T ← SG_T/{T_j} endif
(07) endfor
(08) for each T_j ∈ SG_T do ts ← ts ∪ {< T_j, ⊥ >}; endfor
(09) return ts.
```

Figure 3.4: The createPubTimestamp() function (for a topic manager TM_{T_i}).

We want to remark that an event timestamp has an entry only for those topics that precede T in the topic order (line 03) and that appear in more than one subscription together with T (lines 04-07).

Considering the execution depicted in Figure 3.3 and the topic order $T1 \rightarrow T2 \rightarrow T3$, we show how the timestamp of the event e published on $T2$ is created. The procedure can be summarized in the following steps: (i) $SG_T = \{T1, T2, T3\}$ initially represents the sorted union of all the subscriptions containing T_2 (lines 01-02), (ii) $SG_T = \{T1, T2\}$ is the result after filtering out topics following T_2 according to the precedence relation $\rightarrow$ (line 03), and (iii) $ts_e = \{< T1, \perp >, < T2, \perp >\}$ is the empty timestamp built from SG_T.

When an event e is published on a topic T, the publisher p_i executes the algorithm shown in Figure 3.5(a). In particular, it associates to e a unique identifier generated locally (line 01), it puts the event, together with the topic and the corresponding identifier in a buffer (line 02) and sends a CREATE_PUB_TS (id_e, i, T) message to the topic manager TM_T, associated to T (lines 03-04).

Receiving the CREATE_PUB_TS (id_e, i, T) message, TM_T executes the algorithm shown in Figure 3.5(b). In particular, it first creates an empty timestamp ts_e by executing the createPubTimestamp function, it increments its local sequence number (line 02), updates the corresponding entry in ts_e (line 03) and sends a FILL_IN_PUB_TS message containing the timestamp, to the preceding topic manager until ts_e has been completed and it is finally returned to the publisher. Note that, when a topic manager receives a FILL_IN_PUB_TS message, it just attaches its local sequence number (line 12). Figure 3.3 shows an example of the complete publish procedures for three different events with the corresponding timestamps.

```
operation PUBLISH(e, T):

(01)  id_e ← generateUniqueEventID (e);
(02)  outgoingEvents ← (e, id_e, T);
(03)  dest ← getTopicRespAddress(T);
(04)  send CREATE_PUB_TS (id_e, i, T) to dest;

(05)  when EVENT_TS (ts, e_id) is delivered:
(06)      let < e, id_e, T > ∈ outgoingEvents
(07)      such that (e_id = id_e);
(08)      ENSpublish(< e, ts >, T);
(09)      outgoingEvents ← outgoingEvents/{< e, id_e, T >}.
```

(a) Publisher Protocol (for a publisher process p_i)

```
(01)  when CREATE_PUB_TS (e_id, j, T) is delivered:
(02)      ev_ts ← createPubTimestamp(T, externalSubs_{T_i});
(03)      LC_{T_i} ← LC_{T_i} + 1;
(04)      ev_ts ← update(ev_ts, < T_i, LC_{T_i} >);
(05)      if (|ev_ts| = 1)
(06)        then send EVENT_TS (ev_ts, e_id) to p_j;
(07)        else t' ← next(ev_ts, T_i);
(08)             dest ← getTopicRespAddress(t');
(09)             send FILL_IN_PUB_TS (ev_ts, e_id, j) to dest;
(10)      endif

(11)  when FILL_IN_PUB_TS (ts, e_id, j) is delivered:
(12)      ts ← update(ts, < T_i, LC_{T_i} >);
(13)      t' ← next(ts, T_i);
(14)      if (t' = null)
(15)        then send EVENT_TS (ts, e_id) to p_j;
(16)        else dest ← getTopicRespAddress(t');
(17)             send FILL_IN_PUB_TS (ts, e_id, j) to dest;
(18)      endif.
```

(b) TM Protocol (for a topic manager TM_{T_i})

Figure 3.5: The publish() protocol.

NOTIFY() **Operation.** When an event e is notified by the ENS, a subscriber s_i executes the algorithm shown in Figure 3.6. The algorithm uses a function tag(e) that creates a new event e', containing e and the indication that it has

been delivered out-of-order. The event e is not immediately notified to the application layer. A subscriber s_i first checks if the event has been published on a topic actually subscribed by s_i and then checks if it has been notified by the ENS in the right order (line 01). If such condition is not satisfied, a new event e' is created by tagging e; then, the event e' is notified to the application (lines 09-11).

```
upon ENSnotify(< e, ts >, T):

(01)  if ((T ∈ subs_i) ∧ (sub_LC_i < ts))
(02)    then trigger notify (e, T);          % ordered notification
(03)         for each (< T_j, v >∈ sub_LC_i)
(04)              if((∃ < T_j, v' >∈ ts) ∧ (v' > v))
(05)                then sub_LC_i ← sub_LC_i/{< T_j, v >}
(06)                     sub_LC_i ← sub_LC_i ∪ {< T_j, v' >}
(07)              endif
(08)         endfor
(09)    else e' ← tag(e);
(10)         trigger notify (e', T);          % out-of-order notification
(11)  endif
```

Figure 3.6: The notify() protocol (for subscriber s_i).

On the contrary, if the event can be notified, s_i triggers the notification to the application (line 02) and then updates its local subscription clock with the sequence numbers contained in the event timestamp (lines 03-08).

SUBSCRIBE() **and** UNSUBSCRIBE() **Operations.** The algorithm for a SUBSCRIBE() operation is shown in Figure 3.7. To simplify its pseudo-code, in addition to the functions used in the PUBLISH() algorithm, we defined the createSubTimestamp(sub) function, that creates an empty subscription timestamp, i.e., a set of pairs $< T, sn >$ where T is a topic identifier and sn is the sequence number for T, initially set to $\perp$. The subscription timestamp contains a pair for each topic T of a subscription S.

When a subscriber s_i wants to subscribe a new topic T, it executes the algorithm shown in Figure 3.7(a). In particular, it creates an empty subscription timestamp through the createSubTimestamp function (including also topic T), and then it sends a FILL_IN_SUB_TS $(ts, (subs_i \cup \{T\}), id)$ message to fill the timestamp and to forward the new subscription to the topic manager TM_{T_k} responsible for the last topic in the subscription, according to the precedence relation $\rightarrow$ (lines 02-04).

```
operation SUBSCRIBE(T):

(01)  ts ← createSubTimestamp(subs_i ∪ T);
(02)  t' ← next(ts, null);
(03)  dest ← getTopicRespAddress(t');
(04)  send FILL_IN_SUB_TS (ts, (subs_i ∪ {T}), id) to dest;
      ---------------------------------------------------
(05)  when COMPLETED_SUB_VC (ts, s) is delivered:
(06)      sub_LC_i ← ts;
(07)      subs_i ← subs_i ∪ {T};
(08)      ENSsubscribe(T);
```

(a) Subscriber Protocol

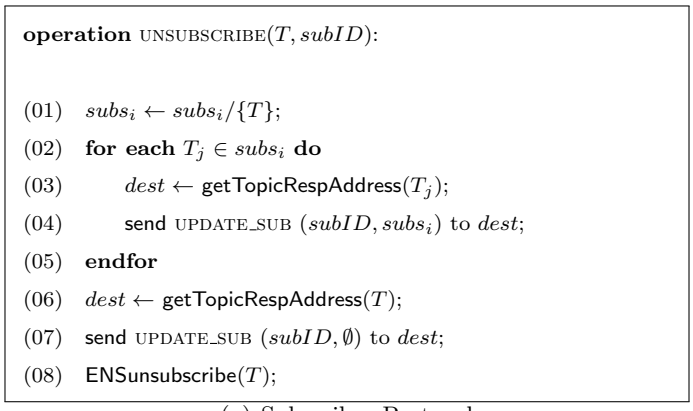
```
(01)  when FILL_IN_SUB_TS (ts, sub, j) is delivered:
(02)      externalSubs_i ← update (externalSubs_i, < j, sub >);
(03)      LC_{T_i} ← LC_{T_i} + 1
(04)      ts ← update (ts, < T_i, LC_{T_i} >);
(05)      t' ← next (ts, T_i);
(06)      if(t' = null) then send COMPLETED_SUB_VC (ts, s) to j;
(07)                    else dest ← getTopicRespAddress(t');
(08)                         send FILL_IN_SUB_TS (ts, s, j) to dest;
(09)      endif
```

(b) TM Protocol

Figure 3.7: The subscribe() protocol.

Upon the delivery of a FILL_IN_SUB_TS message, each topic manager $TM_{T'}$ executes the algorithm shown in Figure 3.7(b). In particular, $TM_{T'}$ updates its $externalSubs_k$ variable with the new subscription (line 02), increments its local sequence number (line 03), updates its entry in the subscription timestamp (line 04) and finally forwards the FILL_IN_SUB_TS message to the preceding topic manager until it is completed and returned to the client. When the subscriber receives the completed subscription timestamp, it updates its local subscription clock (line 06) and then makes the subscription effective by calling the ENSsubscribe() method (line 08).

The algorithm for the UNSUBSCRIBE() operation is shown in Figure 3.8. A subscriber that wants to unsubscribe from a topic T, removes it from the set of subscribed topics (line 01) and, then, informs all topic managers of these topics with the updated subscription through an UPDATE_SUB message (lines 02-05), including the topic manager of T that will receive an empty



operation UNSUBSCRIBE($T, subID$):

(01) $subs_i \leftarrow subs_i / \{T\}$;
(02) **for each** $T_j \in subs_i$ **do**
(03) $dest \leftarrow$ getTopicRespAddress(T_j);
(04) send UPDATE_SUB ($subID, subs_i$) to $dest$;
(05) **endfor**
(06) $dest \leftarrow$ getTopicRespAddress(T);
(07) send UPDATE_SUB ($subID, \emptyset$) to $dest$;
(08) ENSunsubscribe(T);

(a) Subscriber Protocol

when UPDATE_SUB (id, s) is delivered:
(01) **if** ($s \neq \emptyset$)
(02) **then** $externalSubs_i \leftarrow$ update($externalSubs_i, < id, s >$);
(03) **else** $externalSubs_i \leftarrow externalSubs_i / \{< id, - >\}$;
(04) **endif**.

(b) TM Protocol

Figure 3.8: The unsubscribe() protocol.

subscription (lines 06-07). When receiving an UPDATE_SUB message (Figure 3.8(b)), topic managers update the *externalSubs* set accordingly with the received subscription.

3.2.3 Correctness proof

In this Section, we will show that the TNO property holds for any pair of non-tagged events.

Definition 4. *Given a generic subscriber s_i, let us denote $\tau_n(i, e)$ as the time instant at which the ENS notifies an event e to s_i.*

Lemma 1. *Let e_1 and e_2 be two events published respectively on a topic T. If a subscriber s_i notifies e_1 before e_2 then any other subscriber that notifies both e_1 and e_2 will notify e_1 before e_2.*

Proof Let us suppose by contradiction that there exist two subscribers, namely s_i and s_j, that notify both e_1 and e_2 published on topic T, but s_i

notifies e_1 and then e_2 (i.e., $\tau_n(i, e_1) < \tau_n(i, e_2)$), while s_j notifies e_2 and then e_1 (i.e., $\tau_n(j, e_2) < \tau_n(j, e_1)$).

Given a generic subscriber s_k notifying both e_1 and e_2, it follows that at time $\tau_n(k, e_1)$, $T_1 \in S_k$ and at time $\tau_n(k, e_2)$, $T_2 \in S_k$.
Moreover, $sub\_LC_k(\tau_n(k, e_1)) \leq ts_{e_1}$ and $sub\_LC_k(\tau_n(k, e_2)) \leq ts_{e_2}$.
Given the two timestamps ts_{e_1} and ts_{e_2} associated respectively to e_1 and e_2, let us first consider how they have been created and then let us show that it is not possible to have inversions in the notification order.

Let p_k and p_h be respectively the publishers of events e_1 and e_2. When a publisher publishes an event, it executes line 04 of Figure 3.5(a) and it sends a CREATE_PUB_TS message.
Without loss of generality, let us assume that TM_T delivers first the CREATE_PUB_TS message sent by p_k and then the CREATE_PUB_TS message sent by p_h.

Let v be the value of the local clock maintained by TM_T when it delivers the CREATE_PUB_TS message sent by p_k. When TM_T delivers such a message, it creates an empty event timestamp ts_{e_1} through the execution of the createPubTimestamp function (cfr. Figure 3.4), it increments its local clock (i.e. $LC_T = v+1$) and it includes the pair $< T, v+1 >$ in ts_{e_1} (lines 03 - 04, Figure 3.5(b)). Then two cases can happen:

1. ts_{e_1} **contains only the entry for T (line 05)**: $ts_e = \{< T, v+1 >\}$ and TM_T returns the completed timestamp to the publisher for the publication on the ENS.

2. ts_{e_1} **contains more than one entry (lines 06 - 09)**: in this case, there exists a topic T' following T in the topic order and TM_T sends a FILL_IN_PUB_TS message to $TM_{T'}$. Receiving such a message, $TM_{T'}$ just updates the pair $< T', \perp >$ contained in ts_e with its current sequence number for T' and it checks if there exists a topic T'' following T' in the timestamp. If so, it forwards the FILL_IN_PUB_TS message to $TM_{T''}$, otherwise, it returns ts_e to the publisher (lines 11 - 16).

When TM_T delivers the CREATE_PUB_TS message sent by p_h, it repeats the previous actions: it creates a template ts_{e_2} for the timestamp, increments its local sequence number (i.e., $LC_T = v + 2$), includes the pair $< T, v + 2 >$ in ts_{e_2} and sends the timestamp to the publisher or to the following topic manager.

Considering that the subscriptions are stable, the timestamp will always include the same entries, i.e., ts_{e_1} and ts_{e_2} contain a set of pairs differing only for the sequence numbers associated to each topic. In particular, considering that (i) a topic manager can only increment its local sequence number when a publish occurs, and (ii) topic managers are connected through FIFO channels,

it follows that for each topic T_i the sequence number v' associated to T_i in ts_{e_2} cannot be smaller than the one associated to T_i in ts_e. Therefore, $ts_{e_1} < ts_{e_2}$.

Considering that (i) as soon as an event e is notified to the application layer, the local subscription clock of the subscriber is updated according to the event timestamp (lines 03 - 06, Figure 3.6), and that (ii) $ts_{e_1} < ts_{e_2}$, we have that s_j evaluating the notification condition at line 01 will discard event e_1 after the notification of e_1. This clearly leads to a contradiction.

$\square_{Lemma\ 1}$

Lemma 2. *Let e_1 and e_2 be two events published respectively on a topic T_1 and on a topic T_2, with $T_1 \neq T_2$. If a subscriber s_i notifies e_1 before e_2 then any other subscriber that notifies both e_1 and e_2 will notify e_1 before e_2.*

Proof For ease of presentation and without loss of generality, let us assume that T_1 and T_2 are the two only topics subscribed by both s_i and s_j[5] (i.e. $\{T_1, T_2\} \subseteq S_i, S_j$).

Let us suppose by contradiction that there exist two subscribers, namely s_i and s_j, that notify both e_1 (published on topic T_1) and e_2 (published in topic T_2) but s_i notifies e_1 and then e_2 (i.e. $\tau_n(i, e_1) < \tau_n(i, e_2)$) while s_j notifies e_2 and then e_1 (i.e. $\tau_n(j, e_2) < \tau_n(j, e_1)$).

Given a generic subscriber s_k, if it notifies both e_1 and e_2, it follows that, at time $\tau_n(k, e_1)$, $T_1 \in S_k$ and at time $\tau_n(k, e_2)$, $T_2 \in S_k$. Moreover, $sub\_LC_k(\tau_n(k, e_1)) \leq ts_{e_1}$ and $sub\_LC_k(\tau_n(k, e_2)) \leq ts_{e_2}$.

Given the timestamps ts_{e_1} and ts_{e_2} associated respectively to e_1 and e_2, let us first consider how they have been created and then let us show that it is not possible to have inversions in the notification order.

Without loss of generality, let us assume that T_1 has higher priority than T_2 in the topic order (i.e., in any timestamp containing an entry for both T_1 and T_2, T_1 follows T_2 in the sequence). Considering the createPubTimestamp function shown in Figure 3.4, each event published on T_1 will have attached a timestamp containing a pair $< T_1, v_1 >$ and each event published on T_2 will have attached a timestamp containing the following pairs $< T_1, v_1 >, < T_2, v_2 >$.

When e_2 is published by the application layer, the publisher sends a CREATE_PUB_TS request for the event timestamp to TM_{T_2} (line 04, Figure 3.5(a)). Receiving such a request, TM_{T_2} executes line 01 of Figure 3.5(b) (i.e. createPubTimestamp function) and creates an empty event timestamp containing an entry both for T_1 and T_2 (i.e., $ts_{e_2} \supseteq < T_1, \perp >, < T_2, \perp >$), it increments

[5]The proof can be easily extended to multiple intersections, by iterating the reasoning for any pair of topics that appears in more than one subscription.

its local clock, let's say to a value v (line 02), it updates its component of the timestamp with its local clock (i.e. $ts_{e_2} \supseteq < T_1, \perp >, < T_2, v >$) and then it sends a FILL_IN_PUB_TS message containing ts_{e_2} to the following topic manager selected in the event timestamp according to the precedence relation $\rightarrow$ (i.e., to TM_{T_1}).
The same procedure is executed when e_1 is published.

Note that, in the worse case scenario, due to concurrency in the timestamp creation procedure, TM_{T_1} can either deliver first the FILL_IN_PUB_TS message sent by TM_{T_2} and then the CREATE_PUB_TS sent from the publisher of e_2 or viceversa, it can first manage the CREATE_PUB_TS message and then the FILL_IN_PUB_TS message.

1. TM_{T_1} **delivers the** CREATE_PUB_TS **message for event** e_1 **and then the** FILL_IN_PUB_TS **message for** ts_{e_2}. Delivering the CREATE_PUB_TS for event e_1, TM_{T_1} creates an empty event timestamp for e_1 (i.e., $ts_{e_1} \supseteq < T_1, \perp >$), it updates its local clock to $v_1 + 1$, it updates its component of the timestamp with its local clock (i.e., $ts_{e_1} \supseteq < T_1, v_1 + 1 >$) and then it sends a FILL_IN_PUB_TS request containing ts_{e_1} to the following topic manager in the topic order (if any) or directly to the publisher.
   Delivering the FILL_IN_PUB_TS message for ts_{e_2}, TM_{T_1} executes line 11 of Figure 3.5(b), and updates its component of the timestamp with its local clock (i.e., $ts_{e_2} \supseteq < T_1, v_1 + 1 >, < T_2, v >$). Then, it sends a FILL_IN_PUB_TS request containing ts_{e_2} to the following topic manager in the topic order (if any) or directly to the publisher.

2. TM_{T_1} **delivers the** FILL_IN_PUB_TS **message for** ts_{e_2} **and then the** CREATE_PUB_TS **message for event** e_1. Delivering the the FILL_IN_PUB_TS message for ts_{e_2}, TM_{T_1} executes line 11 of Figure 3.5(b), and updates its component of the timestamp with its local clock (i.e., $ts_{e_2} \supseteq < T_1, v_1 > , < T_2, v >$). Then, it sends a FILL_IN_PUB_TS request containing ts_{e_2} to the following topic manager in the topic order. On the contrary, delivering the CREATE_PUB_TS for event e_1, TM_{T_1} creates the template for the event timestamp (i.e., $ts_{e_1} \supseteq < T_1, \perp >$), updates its local clock to $v_1 + 1$, updates its component of the timestamp with its local clock (i.e., $ts_{e_1} \supseteq < T_1, v_1 + 1 >$) and then it sends a FILL_IN_PUB_TS request containing ts_{e_1} to the following topic manager in the topic order (if any) or directly to the publisher.

Note that, TM_{T_1} only attaches its local clock to the timestamp for e_2 without incrementing it; thus, the two cases can be considered together. Let us now consider the behavior of s_i and s_j when the notification is triggered by

the ENS.

Subscriber s_i. At time $\tau_n(i, e_1)$, s_i notifies e_1 and then updates its local clock by executing lines 03 - 08 of the notification procedure. In particular, s_i updates sub_LC_i with the pair $< T_1, v_1 >$ (or $< T_1, v_1 + 1 >$). At time $\tau_n(i, e_2)$, s_i is notified by the ENS about e_2. Since it has updated only the sub_LC_i entry corresponding to e_1, and considering that the value v has been assigned to T_2 for e_2, it means that $sub\_LC_i \leq ts_{e_2}$ and also e_2 can be notified.

Subscriber s_j. At time $\tau_n(j, e_2)$, s_j notifies e_2 and then updates its local clock by executing lines 03 - 08 of the notification procedure. In particular, s_j updates sub_LC_i with the pairs $< T_1, v_1 >$ (or $< T_1, v_1 + 1 >$) and $< T_2, v >$. At time $\tau_n(j, e_2)$, s_j is notified by the ENS about e_2. Looking to ts_{e_2}, it is not smaller equal than sub_LC_i (i.e., there not exists a component where the timestamp is strictly greater than the local clock). Therefore the condition at line 01 is not satisfied, e_1 is discarded and we have a contradiction.

$\square_{Lemma\ 2}$

Theorem 1. *Let e_k and e_h be two events. If a subscriber s_i notifies first e_k and then e_h, any other subscriber that notifies both e_k and e_h will notify e_k before than e_h.*

Proof The proof trivially follows from Lemma 1 and Lemma 2. $\square_{Theorem\ 1}$

3.3 Causality relation among events

In this Section we show that the events published on the same topic maintain a causality relation among them. To this end, we assume that each process that is publisher on a topic T, it is also a subscriber of the same topic. Thus, we define a *causality relation* " $\mapsto$ " as follows:

Definition 5. *If $publish(e) \mapsto publish(e')$, where e, e' are published on the same topic T and e, e' have the same destination, then e has to be notified before e'.*

Note that this definition meets the original definition of causal ordering given by Birman and Joseph in the context of broadcast communications [30] when considering a publish/subscribe system with a single topic. To demonstrate that in our total ordering algorithm events published on the same topic

also respect a causal order, we have to prove that: (i) the relation " $\mapsto$ " follows the *happened-before* relation introduced by Lamport in [76]; (ii) if the relation " $\mapsto$ " is verified, then a subscriber that receives e, e' notifies e before e'. To this end, we introduce the following Lemma:

Lemma 3. *Given two events e, e' published on a topic T, $publish(e) \mapsto publish(e')$ if and only if:*

1. *e, e' are published by the same process, or*
2. *e is published by a process p, while e' is published by a process $q \neq p$ after it has notified e, or*
3. *there exists $publish(e'')$ such that: $publish(e) \mapsto publish(e'') \mapsto publish(e')$.*

Proof For the condition 1, let us assume by absurd that two events e, e', published in this order by the same process p, have the sequence numbers sn_e and $sn_{e'}$ such that $sn_{e'} < sn_e$. Because the sequence numbers are assigned by the same topic manager TM_T and we have assumed the presence of FIFO channels, TM_T will receive e, e' in the same order they have been published by process p. Due to the increasing monotonicity of the sequence numbers assigned by the same topic manager, the event e' cannot have a sequence number lower than the one assigned to e. This obviously contradicts the thesis.

For condition 2, instead, let us assume by absurd that the event e' published by a process q has a sequence number lower than the one assigned to the event e previously published by a process p, with $p \neq q$. Because these sequence numbers are assigned by the same topic manager TM_T, and the event e has been already notified by process q before publishing e', TM_T will receive the event e' after that a sequence number for e has been assigned. Due to the increasing mononicity of sequence numbers assigned by a topic manager, e' cannot have a sequence number lower that e. This clearly leads to a contradiction.

The condition 3 simply follows from conditions 1 and 2 by considering two couples of publish operations, i.e., $publish(e)$, $publish(e'')$ and $publish(e'')$, $publish(e')$. $\square_{Lemma\ 3}$

The next step is to demonstrate that if the relation " $\mapsto$ " is verified, then a subscriber that receives e and e' notifies e before e'. To this end, we

introduce the following theorem:

Theorem 2. *If* $publish(e) \mapsto publish(e')$*, with* e, e' *published on the same topic* T*, then* $ts_e < ts_{e'}$ *and a subscriber that receives both* e *and* e' *delivers* e *before* e'*.*

Proof By the design of our algorithm, a process p subscribed to a topic T that receives an event published on T, will also receive a timestamp that will contain all pairs $< T_i, sn_i >$ related to the topics T_i in the sequencing group of T. Consider two new events e, e' published in this order on T, and let us assume by absurd that the process p receives the timestamps associated to e and e' such that: $ts_{e'} < ts_e$.

From Lemma 3, we have that $sn_e < sn_{e'}$; therefore, e and e' have to be notified in the same order they are published. By considering the whole set of pairs $< T_i, sn_i >$ contained in the timestamps associated to e and e', we have from Lemma 3 that no two events e_i and $e_{i'}$, that could have been published in this order on a same topic T_i in between $publish(e)$ and $publish(e')$, have their sequence numbers such that $sn_{e'_i} < sn_{e_i}$. As such, $ts_e < ts_{e'}$, that contradicts the thesis. $\square_{Theorem\ 2}$

3.4 Engineering aspects

Event Buffering. The algorithm introduced in Section 3.2.2 assumes that received events with old timestamps are tagged to indicate that they are notified out-of-order. The main source of out-of-order notifications lies in the fact that two events, possibly published by different publishers, can follow distinct paths through the ENS, before reaching the point where they will be notified to the final recipients. To reduce the number of out-of-order notifications, we can use a buffering strategy on the subscriber side. Every time the ENSnotify() primitive returns a new event e, the algorithm checks through the attached timestamp whether some other event can exist with a smaller timestamp. This check is performed by looking at the sequence numbers included in the timestamp: if the values for all the topics are equal to the corresponding ones stored locally in sub_LC_i, except for the topic where the event has been published, that must have a value greater than the local one by one unit, then no event with a smaller timestamp exists that must be still received by the subscriber, and the received event can thus be delivered.

If there is a possibility that an event with a smaller timestamp exists but has not been delivered to the subscriber so far, then the event e is queued

in a buffer able to host a maximum of b events and a timer for e is started (TTL_e). The event e is delivered through the notify() primitive when one of the following conditions holds: (i) all the events with smaller timestamps have been notified, (ii) TTL_e expires or (iii) the buffer is full, a new event must be buffered and e is at the head of the queue.

Reliability. Making the algorithm presented so far working reliably in an environment where messages can be lost requires some more minor changes. The loss of a message during the timestamp generation phase, for example, could lead a publisher to wait forever before publishing an event in the ENS. This problem can be solved with a simple retransmission approach: the publisher periodically re-initiates the procedure for building the timestamp until it receives a correct timestamp for the event. During the timestamp construction procedure, TMs buffer partially filled-in timestamps and retransmit them as soon as they receive another request for the same timestamp. When a timestamp has been completely filled-in, a message can be routed through the appropriate TMs to free their buffers. Finally, the internal state of TMs should be preserved despite possible process failures in order to avoid possible TNO violations. This can be obtained by adopting standard replication techniques [31, 97].

Dynamic topic ordering. The algorithm described so far assumes a fixed topic ordering that is given and known by all the participants. This ordering has a strong impact on the performance of the algorithm at run-time as it is used to decide the content of each timestamp. Depending on the intersection among subscriptions, and on the topic ordering, the timestamp for an event published on a topic can have different sizes spanning from a single entry, up to an entry for every topic in the system. This size impacts the time needed to build the timestamp as it will travel through all TMs of topic it contains. Ideally, topics where a lot of events are published should thus appear in the highest ranks in the topic ordering such that their timestamp will probably contain less entries. However, accurate statistics on the popularity of topics are not always available at configuration time and, moreover, they only describe statistical properties ignoring transient behaviors that can adversely impact system performance for non negligible time periods. In this Section we describe a topic swapping procedure that modifies the topic ordering adapting it at run-time to the current topic publication popularity.
We assume the presence of a special system topic T_s subscribed by all TMs, which is used to advertise that a new topic swapping procedure is happening. T_s is managed by a TM, say TM_{T_s}, as all other topics in the system. In addition, T_s has an associated sequence number that represents the number of

swaps occurred so far in the system, and it is used to clearly define subsequent *ordering epochs*, i.e. periods of time where different topic orderings are considered. All TMs maintain a local copy of this sequence number in LC_{T_s}.
The topic swapping procedure relies on a function $f()$ that, when applied on a topic T, returns a comparable metric that can be then used by a TM to check if one of the other TMs managing topics with lower priorities (i.e. a topic T' such that $T \rightarrow T'$) in the topic order is a candidate for swapping. In this case a swapping procedure takes place: when TM_T wants to swap position with $TM_{T'}$, it contacts TM_{T_s} and communicates that T has to be exchanged with T' in the topic order. Then, TM_{T_s} increments LC_{T_s} and inserts it in a message together with the new topic order. This message is sent to the TM with lower priority in the topic order and will traverse all the TMs; at the end of the procedure each TM will be informed about the swap and will update the topic order and the value of LC_{T_s}. This value determines a new epoch: when a TM receives a timestamp with a previous sequence number for T_s, it simply discards that message. The definition of function $f()$ is tied to the application; from a general point of view, it should take into account the topic publication popularity as this metric can lead to shorter timestamps. However, other aspects can be considered as well. For example $f()$ could be structured in order to push topics associated to TMs with more available resources (networking and computational) toward higher priorities where larger loads are incurred.

Dynamic reordering: correctness proof. Let us consider two topic managers TM_{T_1} and TM_{T_2} respectively of topics T_1 and T_2, currently in the order $T_1 \rightarrow T_2$ and with a higher publication rate on T_2. TM_{T_1} notices that $f(T_2) > f(T_1)$; thus it contacts TM_{T_s} for a new swap. TM_{T_s}, in turn, increments LC_{T_s} by one and inserts this value in a message containing the new ordering rule $T_2 \rightarrow T_1$. The message is forwarded along the ordered sequence including all TMs and it is returned to TM_{T_s}. If it does not receive back the message within a timeout expiration, TM_{T_s} assumes that the message has been lost and resends it. Thus, eventually all TMs will receive the message and update both the order of TMs in the chain and the local copy of LC_{T_s}. All events published in the system have a timestamp containing a sequence number for T_s: in this way all timestamps received by a subscriber have at least one entry in common and can be *comparable*. When a TM receives a message with a previous sequence number for T_s, it simply discards that message. It is worth mentioning that T_s is not involved in the swapping procedures: it can be always considered as the last topic in the sequence.

During the topic swapping procedure, there is a transitory phase during which some TMs could not be notified yet about a swap. Consider the topic sequence $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$ and two distincts ordering paths $T_1 \rightarrow$

$T_2 \rightarrow T_5$ and $T_1 \rightarrow T_2 \rightarrow T_4$. At time t, TM_{T_1} contacts TM_{T_s} for a topic swapping between T_1 and T_2. Thus, TM_{T_s} increments LC_{Tm} and inserts it in a message together with the new order $T_2 \rightarrow T_1$. This message will be delivered by TM_{T_5}, TM_{T_4}, TM_{T_3}, TM_{T_2}, TM_{T_1} in this order. Suppose that at time $t' > t$ TM_{T_5} creates a timestamp for a new publication on T_5: this timestamp will include T_1 and T_2 other than T_5. We can distinguish between two cases:

1. TM_{T_5} did no receive the message from TM_{T_s}, thus the new timestamp contains $< TM_{T_1}, TM_{T_2}, TM_{T_5} >$ in this order. Because at time t T_5 preceeds T_1 and T_2 in the chain and TM_{T_5} did not received the message from TM_{T_s}, TM_{T_1} and TM_{T_2} did not received that message as well. The execution proceeds seamlessly as before time t (i.e., before topic swapping).

2. TM_{T_5} received the message from TM_{T_s}, thus the new timestamp contains $< TM_{T_2}, TM_{T_1}, TM_{T_5} >$ in this order. Two subcases can verify:

   - TM_{T_1} and TM_{T_2} received the message from TM_{T_s}, thus all TMs along this ordering path have updated information about the new topic order.
   - At least one between TM_{T_1} and TM_{T_2} did not received the message from TM_{T_s}, for example TM_{T_1}. However, TM_{T_1} can verify that the sequence number for T_s contained in the timestamp created by TM_{T_5} has been incremented with respect to its local copy, and from the timestamp itself it can infer the new topic ordering rule. Thus, TM_{T_1} updates information about LC_{T_s} and the new topic ordering.

Finally, let us suppose that TM_{T_4} did not received the message from TM_{T_s} and creates a timestamp for a new publication on T_4 at a time $t'' > t'$, with TM_{T_1} and TM_{T_2} that have updated their information based on the timestamp previously created by TM_{T_5}. When TM_{T_1} receives the timestamp created by TM_{T_4}, it notices a previous sequence number for T_s and, therefore, discards that message.

3.5 Performance Evaluation

In this Section we evaluate the behavior of our ordering module implementing the proposed algorithm. In this evaluation we use SCRIBE as the underlying ENS, and Pastry [96] as a point-to-point communication substrate. We further assume that subscriber and publisher roles are played by the same nodes that constitute the ENS.

We first show how ordering affects the system performance in a large scale environment through a simulation-based study. In such a scenario we also show how it is possible to reduce the impact of ordering through our dynamic topic adaptation based on publication popularity. In addition, we also present how this mechanism is able to adapt publication popularity even when this popularity changes. Then, in a second phase we show the performance of the real prototype we implemented in a small scale; in particular, we provide a comparison between our algorithm and a solution based on JGroups, a toolkit for reliable and ordered multicast communication. This study assesses that with a low/medium number of published events per time unit the performance of the two algorithms are very close, while for a higher publication rate our solution outperforms the one based on JGroups.

3.5.1 Settings and metrics

We used *FreePastry* [57] to implement and evaluate our ordering algorithm. *FreePastry* is a Java tool that provides both a simulator and a prototype of SCRIBE and Pastry.

In the simulated setting, we considered an underlying physical network characterized by two channels types [17]: *fast channels* for short/medium distance (80% of all links) and *slow channels* for long distance (20% of all links). Both were modeled by a Gaussian distribution with mean latency 21ms and 240ms, and standard deviation 10.85ms and 129.27ms respectively. This carachterization allows us to model the whole SoS as the interconnection of systems deployed in national landscapes by means of long distance overlay channels. Following realistic implementations of the applications described in Section 1.2, we assume that publishers and subscribers are deployed in all national systems.
In the prototype-based setting, we deployed our algorithm on virtual machines hosted in 2.8 GHz quad-core dual processor physical machines interconnected with 10Gbps network links. Each virtual machine was equipped with 1 GB of RAM and Linux Ubuntu 10.4 as the OS. We used the WANem [110] network emulator to emulate links with standard ADSL bandwidth and an average latency of $21ms$ in order to emulate the behavior of small/medium scale WAN.

The following metrics have been considered:

End to end latency: represents the time taken by an event for traveling from the publisher to the last notified subscriber and it is measured in seconds. In our experimental analysis we separately tracked for each event the time needed for building the timestamp and the time taken by SCRIBE to

notify all the intended subscribers.
Percentage of tagged messages: represents the percentage of messages that the ordering module tags because they have been received out-of-order.
Percentage of notifications: represents the ratio between the number of events published and notified to all interested subscribers in a second.
Bandwidth overhead: represents the additional bandwidth usage imposed by timestamps and it is measured in bytes. In particular, each timestamp entry $< T_i, sn_i >$ measures 24 bytes: we consider the topic identifier T_i as a Pastry object identifier (i.e., 16 bytes), and the sequence number sn_i as a Java *long int* (i.e., 8 bytes).
Message overhead: represents the load imposed by our algorithm on topic managers during the construction of timestamps. Specifically, it is the ratio between the number of messages processed by a topic manager to build timestamps for events published on topics managed by other nodes, and the total number of processed messages during the construction of timestamps.
Percentage of sequence detection: represents the ratio between the number of event sequences correctly detected by all subscribers and the number of event sequences occurred during an execution of the algorithm (despite how many subscribers detected them).

All the values reported in the following are the result of at least 10 independent runs (we did not observe standard deviation above 5% of reported values, thus they are not plotted on the curves).

Parameters we vary in our analysis are:

Event rate: number of events published per second.
ENS size: number of processes in the system.
Buffer size: maximum number of messages temporarily stored in the buffer on the subscriber side.
Number of topics: number of topics in the system available for subscriptions and publications.
Number of subscription: number of topics subscribed by each subscriber.
Publication model: we model publications as a probability distribution over the set of topics. We consider random uniform distribution or power-law distribution with shapes 0.269 and 0.901. These two values respectively refer to the 40% and 0.5% of topics having a probability of 80% to be selected for a new publication. Moreover, we consider an additional worst case scenario that consists in publishing always on the last topic in the topic order; this represents a disadvantageous scenario for our algorithm as building timestamps will require messages to travel through a long list of TMs.
Subscription model: as for publications, subscriptions are modeled as a

probability distribution over the set of topics. Again, we consider random uniform distribution or power-law distribution with shapes 0.269 and 0.901. Moreover, we consider an additional scenario in which subscribers subscribe all topics; this represents a particular scenario where all subscribers are part of a single group where all published events are notified (typical setting for broadcast protocols).

3.5.2 Simulation results

In this Section, we first analyze performance assuming a given static topic ordering and we show how results are strongly influenced by this order, then we switch to a setting where dynamic adaptation is enabled and we show the performance improvements obtainable with the topic swapping procedure. Finally, we also show how the presented ordering algorithm helps in augmenting the percentage of event sequences detected by two different subscribers.

Static topic ordering. First, we measure the mean end to end latency for event notification by considering both the time spent for timestamp generation and for event diffusion and notification, varying the ENS size in the range [10-10000]. We consider a scenario with 50 topics subscribed by all subscribers; the event rate was set to 1 event/sec and the simulated time is 30 minutes. The results are reported in Figure 3.9.

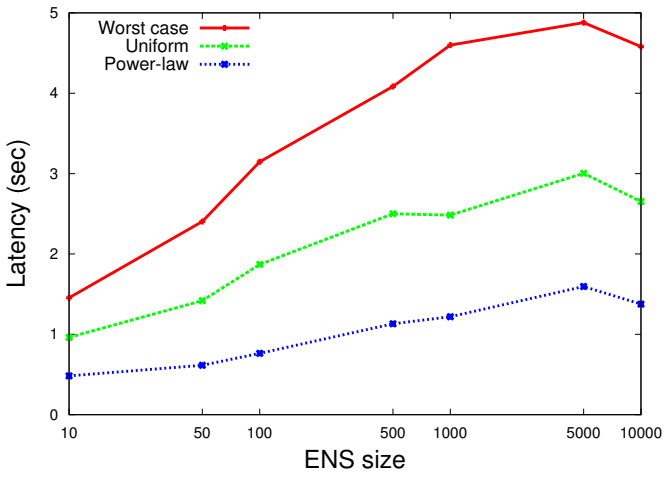


Figure 3.9: End to end latency for event notification vs ENS size with different publication models.

Each curve refers to a different publication model: worst case, uniform distribution and power-law distribution with shape 0.901. In this last case,

the power-law distribution selects more frequently topics at the beginning of the topic order. The curves show the strong impact that different publication models have on notification latency. The coupling between the worst case publication model and the fact that all subscribers subscribe all topics means that the timestamp will travel through all the 50 *TM*s during the generation phase before returning to the publisher and this clearly has a negative effect on the latency that steeply grows with the ENS size. Conversely, in a more favorable scenario where events are published more often on topics with higher priority (power-law model), the latency increment remains reasonable despite the system growth.

In Figures 3.10 and 3.11 we evaluate separately the time spent for timestamp generation and for event diffusion and notification, considering worst case and power-law publication models respectively. These curves clearly show the impact of the ordering algorithm on latency: it is comparable to event diffusion latency for the power-law model, but it completely drives the overall latency with the worst case model. These curves highlight that our algorithm has a non negligible impact on the event diffusion latency, but this impact can be drastically reduced as long as the topic order is carefully chosen to match the publications popularity.

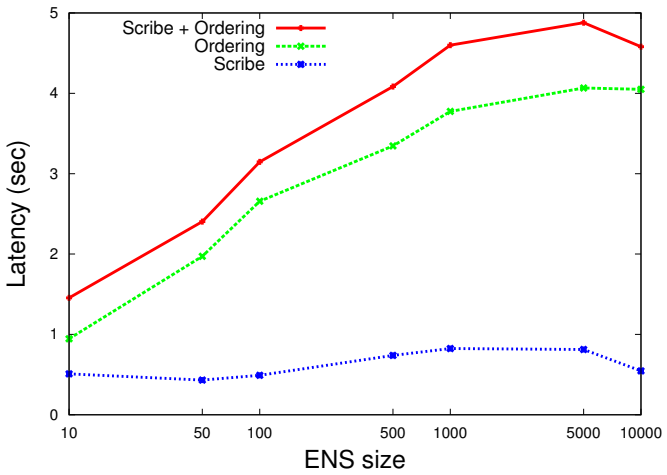


Figure 3.10: Differentiated ordering and ENS latencies vs ENS size with a worst case publication model.

The same consideration can be inferred by looking at Figure 3.12, that shows the percentage of notifications in a second. Even in this case, an advantageous topic order helps in augmenting the fraction of events notified within a second; this can be particularly useful for high throughput applications.

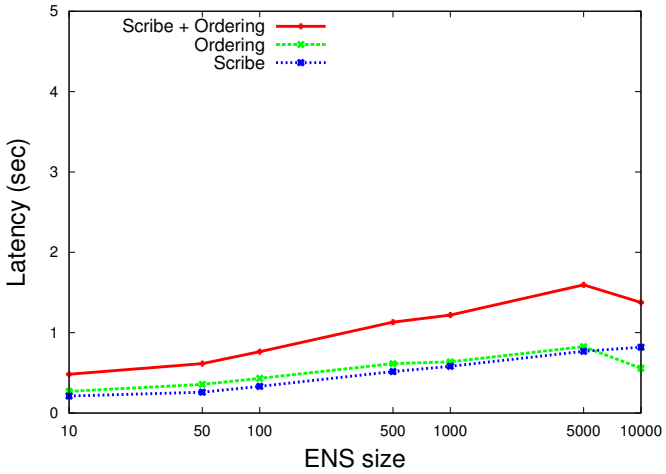


Figure 3.11: Differentiated ordering and ENS latencies vs ENS size with a power-law (shape 0.901) publication model.

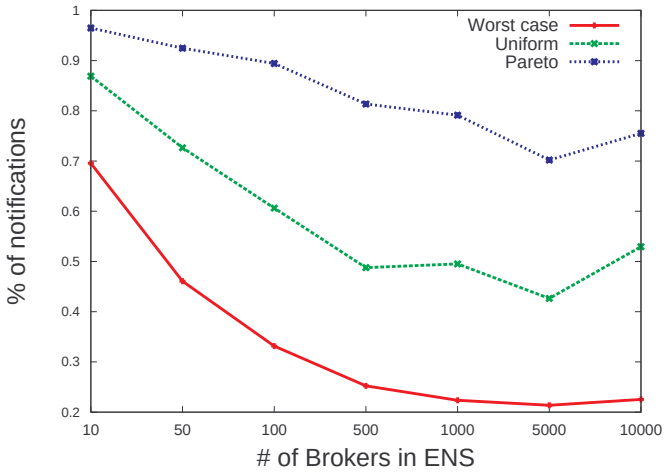


Figure 3.12: Percentage of notifications in a second with different publication models.

Figures 3.13 and 3.14 illustrate the bandwidth overhead imposed by the usage of timestamps and the message overhead due to the timestamps construction by varying the number of subscriptions. In both cases we consider publication and subscription models based on two power-law distributions with shape 0.901, one with the most popular topics at the top of the topic order (referred to as *best case*), and one with the most popular topics at the bottom (referred to as *worst case*). In addition, we also consider publication and subscription models based on a uniform distribution.

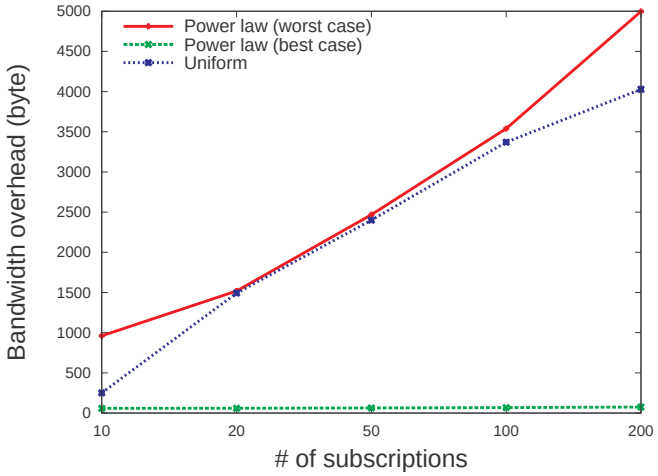


Figure 3.13: Bandwidth overhead due to the usage of timestamps.

The Figures show that when the topic order follows the publication popularity, the overhead imposed by our algorithm both in terms of bandwidth and messages is quite low, and it is not affected by the number of subscription. This is motivated by the fact that in a more *favorable* scenario the timestamp size decreases, and, in turn, the number of processed messages to construct it.

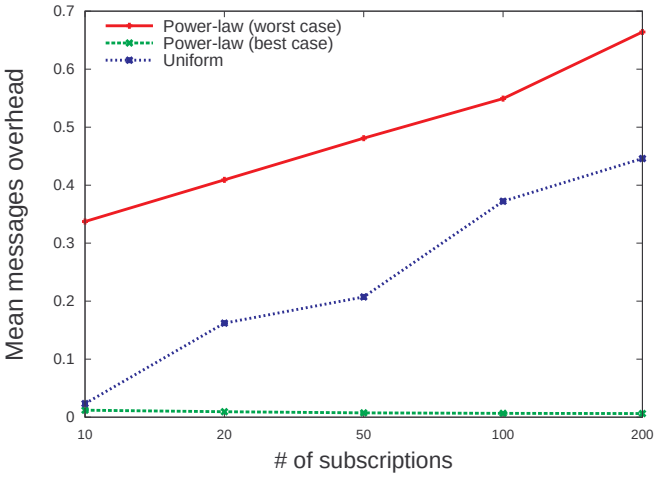


Figure 3.14: Message overhead imposed by the timestamp construction on topic managers.

Finally, we conclude this part by evaluating the trade-off between the percentage of tagged messages and the delivery latency when we vary the size of the buffer used by subscribers. The simulated scenario is the one described above, with publication and subscription models following a power-law distri-

bution with shape 0.901.

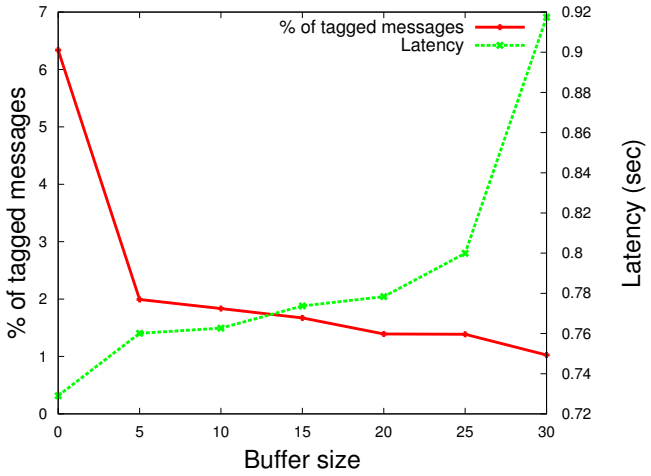


Figure 3.15: Trade-off between notification reliability and latency vs buffer size.

Figure 3.15 highlights how the presence of a buffer helps in augmenting the number of ordered messages delivered to the application, at expenses of an increment of the latency. Applications can decide how to tune the system in order to obtain a configuration that satisfies timeliness requirements and/or helps in delivering a higher number of ordered messages. However, the curves show how just a small buffer can greatly improve this number without impacting too negatively the notification latency.

Dynamic topic ordering. In the previous paragraph we have shown the benefit of configuring the topic order in accordance with the topic publication popularity. In this paragraph we evaluate our dynamic topics adaptation algorithm, which aims to adapt at run-time the topics order to publication popularity. The function we adopted in our experiments was $f(x) = e^{-1/\alpha(x+1)}$ where x is a sequence number. A topic manager TM_{T_i} applies this function on the sequence number sn_i of the topic T_i it manages and on all sequence numbers of other topic managers in the timestamp it receives: if a sequence number sn_j exists such that $f(sn_j) > f(sn_i) + \beta$, the positions of T_i and T_j in the topic order must be swapped. The rationale behind the use of this function is that it eventually converges to $f(x) = 1$; in this way, topics with highest publication rates eventually will reach an almost stable position in the sequence, avoiding swapping procedures that would bring only useless overhead to our algorithm. The parameter α helps in tuning how fast the function convergence is: a smaller value delays this convergence allowing an

higher number of swaps. In this way, the topic sequence quickly adapts to the current publication popularity. However, in order to prevent continuos topics swapping, we allow two topics T_i and T_j to swap their position only if $f(T_j)$ is larger than $f(T_i)$ for a fixed threshold β. In these tests, we considered a setting with 10000 nodes, 1000 topics, 100 topics subscribed per subscriber and event rate fixed at 1 event/sec. In Figures 3.16 and 3.17 show how to tune parameters α and β.

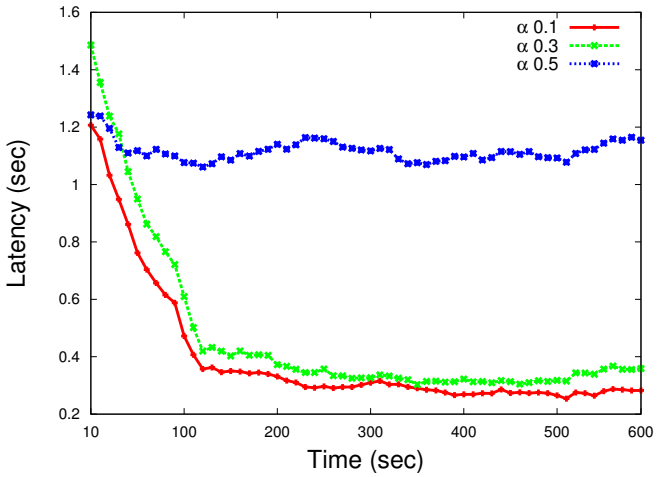


Figure 3.16: Performance of the algorithm with dynamic adaptation for different α values.

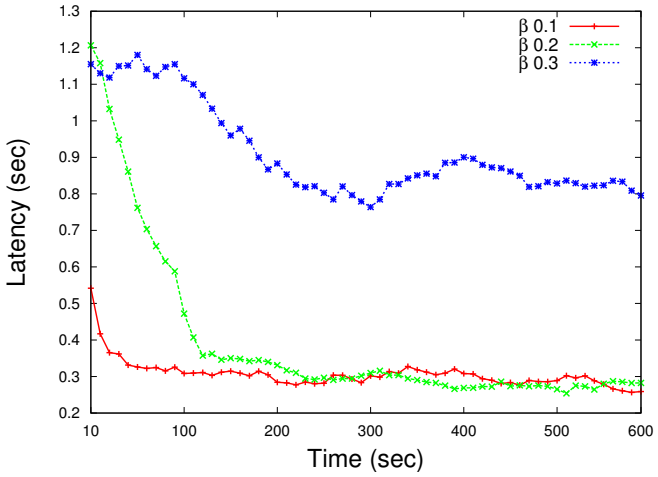


Figure 3.17: Performance of the algorithm with dynamic adaptation for different β values.

Figure 3.16 shows the effect of α on the system, with its value that varies in the set $\{0.1, 0.3, 0.5\}$. While the ordering algorithm converges to a small mean latency with $\alpha = 0.1$ or 0.3, with $\alpha = 0.5$ there is no benefit from dynamic adaptation: in this case function $f()$ grows rapidly allowing few topic swaps. The average number of swaps performed during our tests ranged from 15.1 ($\alpha = 0.1$) to 7.4 ($\alpha = 0.5$). Figure 3.17, instead, shows the effect of β on the system when it varies in the set $\{0.1, 0.2, 0.3\}$. With $\beta = 0.1$ the latency overhead imposed by the ordering algorithm is very low, at the expenses of a high number of swaps (51.8 on average). On the contrary, with $\beta = 0.3$, the algorithm convergence speed is lower, due to a reduced number of swaps (7.8 on average). In this case, dynamic topic ordering has no benefit on the performance of our algorithm. The value $\beta = 0.2$ represents a good trade-off between the two extreme cases: the convergence time of the algorithm to a low latency is still reasonable, with limited overhead imposed by topic swapping. All the following tests were performed setting $\alpha = 0.1$ and $\beta = 0.2$. Figure 3.18 reports the average end-to-end latency for event notification as the simulation evolves in time.

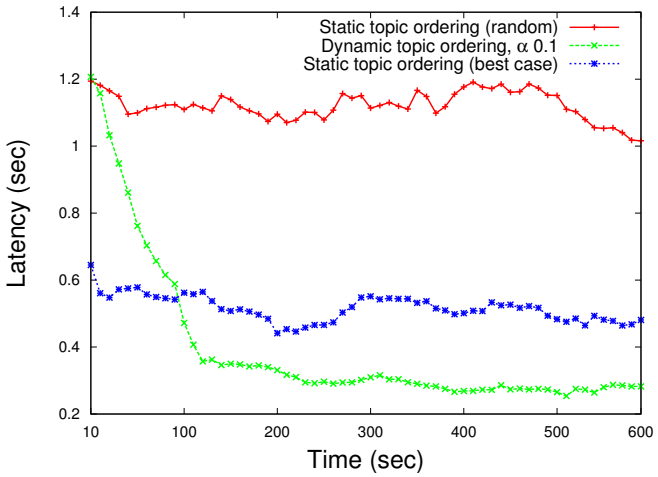


Figure 3.18: Comparison between the algorithm with dynamic adaptation and the static one (best case, and average case).

In this scenario we consider both publications and subscriptions following a power-law distribution with shape 0.901. The *best case* static topic ordering curve assumes that the initial topic order follows the publication popularity distribution. Conversely, the *random* static topic ordering and the dynamic topic ordering curves assume that the initial topic order is randomly chosen. Each point in the picture represents the average notification latency for 10 published events. The curves clearly show how dynamic adaptation allows the

ordering algorithm to quickly converge to a small average latency even if the starting topic order was random. The interesting aspect is that the dynamic adaptation outperforms the *best case* static ordering: indeed, while topic order in the latter case is decided only on the basis of the statistical properties of the publication popularity distribution, dynamic adaptation is able to tune topic order following the real distribution of publications happening at run-time, thus taking into account also possible temporary fluctuations from the statistical properties of their distribution. In addition, the dynamic topic re-ordering procedure produces an improvement also in the bandwidth overhead: Figure 3.19 shows that, with time, due to the reduction of the timestamps size, this overhead is drastically reduced, converging to a mean value of about 100 bytes.

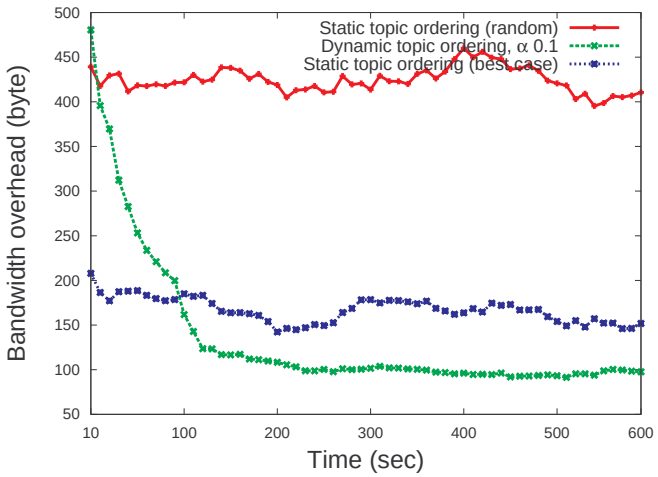


Figure 3.19: Comparison between the bandwidth overhead in the static and dynamic scanarios.

Finally, we show the behavior of the ordering algorithm in a special setting where the publication popularity distribution is abruptly changed at run-time. Figure 3.20 depicts a scenario in which at time 300 sec. we shuffle the topics list in order to modify the frequency at which topics are returned by the power-law distribution used to model publications. In correspondence of this popularity change, the average latency has a steep increase justified by the fact that the topic order to which the algorithm converged so far is no more the best one for the new publication popularity distribution. However, the dynamic adaptation procedure is able to quickly converge back to a new stable topic ordering that brings back performance in terms of latency to the values shown in the previous tests.

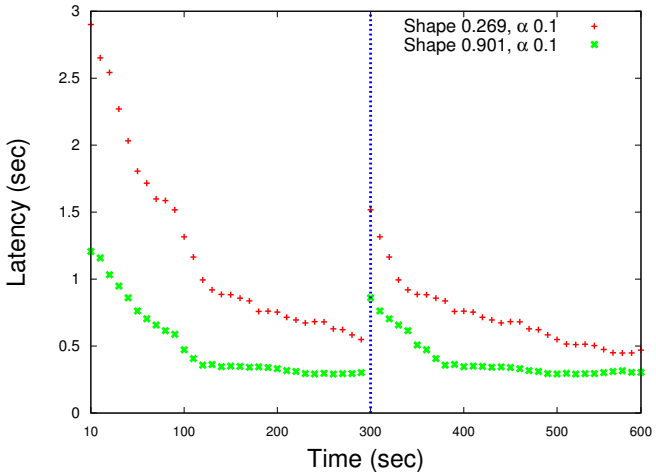


Figure 3.20: Behaviour of the algorithm with dynamic adaptation with an abrupt change in the system configuration.

Impact of notification order on event pattern detection. In this test we try to recreate a scenario similar to the one that can be encountered in composite event pattern detection applications. We consider two subscribers that have to detect the pattern *"event e precedes e′, that in turn precedes e″"*. The system is composed by five publishers that publish on five different topics, two subscribers acting as pattern detectors and an ENS populated with 100 nodes. Each publisher publishes on its topic one of the three events chosen at random at a rate of 5 events/sec. The values reported in Figure 3.21 show the percentage of sequence detections for different settings. The leftmost bar refers to the bare-bone ENS without ordering: in this case the correct detection of the searched pattern is driven only by chance as the ENS will not enforce any kind of ordering. As the result shows, only 35% of the patterns are detected by both subscribers.

The addition of our algorithm to the game completely changes the Figures: the rightmost three bars show the percentage of consistently detected patterns when the algorithm is configured with buffer 0, with buffer 30 and with infinite buffer respectively. In particular, the difference between the bare-bone ENS without ordering and the ENS with the addition of our algorithm, with buffer 0, lies in the fact that the ordering protocol forces events to pass through the network of topic managers, that impose an order among those events. However, several events can be notified out-of-order to different subscribers; hence, the presence of the buffer is not only useful for minimizing out-of-order notifications, but it is also necessary for sequence pattern detection. In fact,

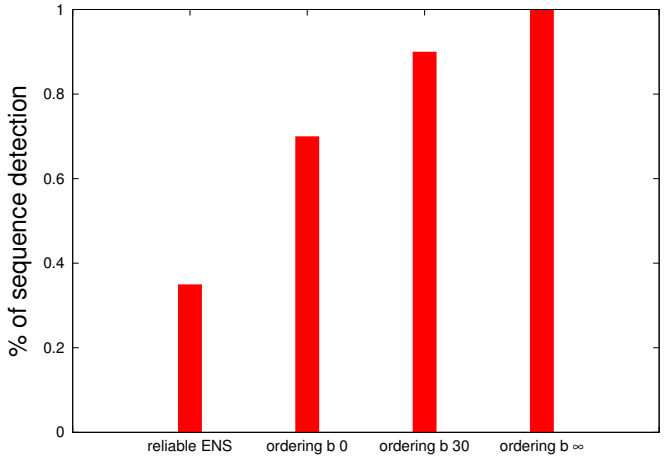


Figure 3.21: Percentage of consistent discoveries of sequence patterns at two different event engines.

note how, in the case with infinite buffer, all the patterns are consistently detected by both subscribers that can thus produce consistent outputs.

Prototype performance evaluation. In this Section, we evaluate the performance of our prototype, also comparing the described ordering algorithm with a solution based on the JGroups toolkit. In all our experiments we varied the ENS size from 1 to 10 nodes. In addition, we have a single publisher that can publish on 5 different topics according to the *publication model*, while subscribers are subscribed to all topics. The obtained results are the average of 5 experiments of 5 minutes each.

We first evaluated the *end to end notification latency* by varying the *ENS size* and the *publication model*, while the *event rate* is fixed to 1 event/sec. Figure 3.22 shows that an increase of the number of nodes in the ENS causes a corresponding increase of the latency. However, the latency quickly reaches a stable point, as its value is mainly driven by the timestamps length. In presence of a single node, the whole system collapses in a centralized sequencer that timestamps and distributes each incoming event.

In Figure 3.23 we evaluated the *percentage of notifications* by varying the *event rate* in the range [5-100] and the *publication model*, while the *ENS size* is fixed to 10. The Figure shows that the overhead imposed by the ordering algorithm in terms of latency may have an impact also on the percentage of the events notified in a second. This can be particularly disadvantageous in high throughput applications. However, in presence of a more advantadgeous

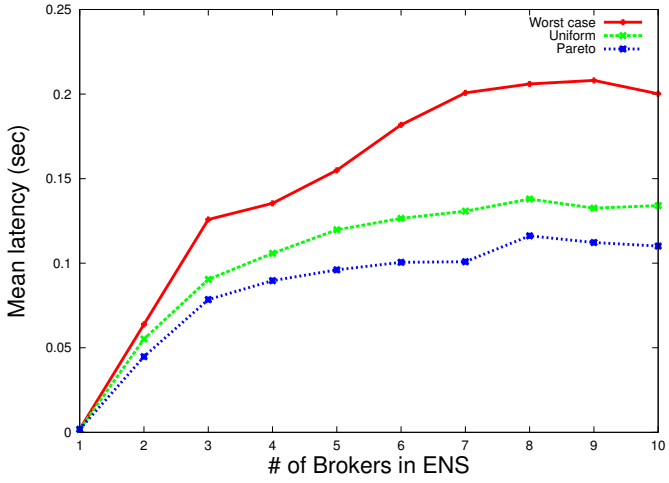


Figure 3.22: End-to-end notification latency in a real WAN setting.

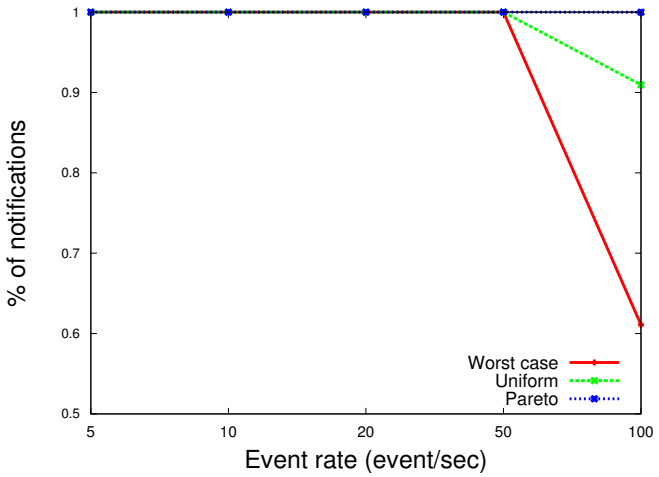


Figure 3.23: Percentage of notifications in a second by varying the event rate.

subscription model, the impact of the latency overhead can be drastically reduced, without affecting the number of notifications per second.

Finally, we provide a comparison between our ordering algorithm and a similar application developed on top of JGroups. Ordered delivery is guaranteed only within a group, thus we developed an application that simply publishes all events in a single group (independently from the topics). Sub-

scribers can then discard upon delivery events published on topics they are not subscribed to.

In our experiment we assumed that no subscriptions or unsubscriptions are issued. JGroups was configured to use the SEQUENCER stack protocol. The usage of the WANem network emulator forced JGroups to use TCP point-to-point channels instead of more performing primitives (e.g. UDP-based IP Multicast). For the ordering algorithm the setting were similar to those described above, with the *ENS size* set to 10 nodes and the *event rate* varying in the range [5 - 100] events/sec.

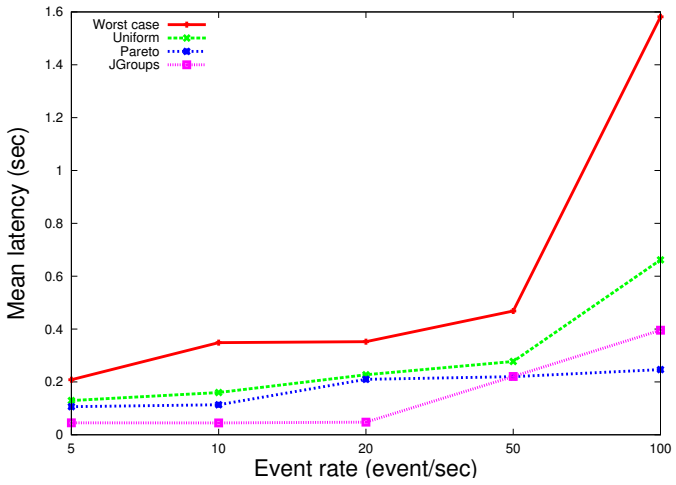


Figure 3.24: Comparison between our algorithm and a solution based on JGroups. With a high event rate our solution outperforms the one based on JGroups.

Results in Figure 3.24 show how our algorithm delivers performance close to JGroups in terms of mean latency as long as a favorable topic ordering is chosen. When the event rate grows our solution starts to outperform JGroups (see the curve related to the power-law publication model); this is a consequence of two different aspects: while on one hand the impact of the ordering algorithm is limited as timestamps are maintained small, on the other hand the application-level tree-based multicast strategy employed by SCRIBE delivers better performance with respect to the point-to-point based multicast strategy employed by JGroups.

3.6 Related work

Totally ordered communications. Totally ordered communications have been extensively studied in the literature and there exists a considerable amount of work on total order broadcast primitives following different approaches. A common point is represented by the need of a certain degree of synchronization and knowledge of the system. As an example, [58] is based on a propagation graph to support multiple overlapping groups: authors use a *fixed sequencer* approach in which sequencers are intermediary nodes of the graph placed at the intersection of different groups and messages are propagated through a series of sequencers that order them by merging messages destined to different groups. Differently, Gopal and Toueg in [61] use a *token-based* approach in which the execution of processes is synchronized according to rounds. Lamport in [76] uses a *communication history* approach: messages carry a timestamp and can be broadcasted at any time. Destinations observe the communication history, i.e., previously generated messages and their timestamps, in order to understand when a message must be delivered to preserve total order. Chandra and Toueg [36] use a *destination agreement* approach in which destinations run a consensus algorithm to agree on a set of messages to deliver. Most of the existing total order broadcast approaches and algorithms are extensively surveyed in [45].

All previous algorithms work properly in a small network while they scale badly with respect to number of participants and their geographical distribution. The sequencer represents a bottleneck as well as a single point of failure. Additionally these algorithms require a certain degree of synchronization among the interacting participants and this is in contrast with basic principles of a publish/subscribe paradigm such as time (i.e., asynchronous notifications) and space decoupling. The ordering mechanism proposed in this paper does not require either any prior knowledge on the system or synchronization among the participants but it relies only on the set of available topics and subscriptions, matching thus the publish/subscribe paradigm principles.

An ordering algorithms for publish/subscribe systems is presented in [82]. Similar to our work, authors use a *sequencing network* to order events across multiple groups of subscribers. However, their solution suffers of two problems: (i) it is not able to handle subscription dynamics, and (ii) a new subscription/unsubscription can create loops in the sequencing network (*circular dependency problem*). In this last case, the sequencing network must be rebuilt from scratch. On the contrary, our solution solves these problems by defining a total order relation among topics that determines a one-way sequence of topic managers that establish an order for events.

Two interesting solutions for ordering in content-based publish/subscribe middleware recently appeared in [118] and [72]. Differently from our work,

authors in [118] ensure total order by also defining a *Uniform Agreement* property: as such, two correct processes interested in two events e and e' both deliver them and they do so in the same order. Each publisher has to *advertise* all brokers about the set of events it will publish. To this end, a broadcast procedure is employed to create a spanning tree rooted at the publisher. Subscriptions are instead forwarded hop-by-hop along the reverse path of matching advertisement trees up to the publisher. While in our solution the correct order of events is reconstructed on subscribers' side, in [118] this task is performed by brokers, that use advertisements and subscriptions to detect a *conflict*, i.e., an out-of-order notification to one or more subscribers. Conflicts are resolved through an acknowledgment mechanism: a broker issues a request to the next hop with conflicting advertisement. The next hop replies back with an ack message if it can determine that there is no conflict. On the other hand, the paper in [72] presents a partition-tolerant content-based publish/subscribe algorithm that can tolerate cuncurrent failures of brokers or communication links up to a value δ. Differently from our solution and the algorithm in [118], the work in [72] ensures a per-source FIFO order.

Finally, authors in [114] investigate the problem of multi-delivery multicast in asynchronous systems in presence of crash-stop failures. They introduce an aggregation model based on a predicate grammar for expressing conjunctions of types of events and properties for the multicast primitives. The paper shows that a total order is necessary to guarantee an agreement on events notified to processes interested in identical conjunctions. In particular, this is shown by deploying an algorithm that implements the described aggregation model on top of a total order broadcast and vice-versa for a majority of correct processes.

Timestamping techniques. Logical clocks have been introduced by Lamport in [76] to identify the causality relation among events of a distributed computation. In [87], the notion of vector clock has been introduced to capture such causality relation. A vector clock is composed of n entries, one for each process in the system while a logical clock is an integer. Logical clocks and vector clocks have been used to solve many basic problems such as transaction management, coordination protocols, ordered communication protocols, message stability protocols, distributed predicate detection just to name a few [20].

At a first glance, timestamps used in our algorithm resemble vector clocks, but they are very different structures. Vector clocks have a well defined and fixed structure that depends on the size of the distributed computation in terms of processes. Therefore the causality relation among two events can be detected just comparing (entry by entry) the two vector clocks associated with the two events. On the contrary, our timestamp structure is independent from the system size but it rather depends on the current set of subscrip-

tions. This is why, the ordering relation among two events can be detected looking, first, to the structure of the timpestamps (i.e., the events have to be comparable according to Definition 3) and secondly, if the timestamps are comparable, the ordering between the two events can be detected examining the values contained in common entries of the timestamps. Let us finally remark that in our timestamping technique, the timestamp associated with an event does not bring any information about the producer of the event, this matches the anonymity principle of a publish/subscribe paradigm (producers and consumers do not know each other).

In some work, events are timestamped with physical clocks. As an example, [79] uses accuracy interval-based timestamps relying on NTP synchronized local clocks as a global time reference. The interesting aspect of this timestamping technique lies on the fact that the order of events follows the real time order. However, such a technique has many drawbacks. Many events could be issued in the same physical time interval by producers (so they have the same physical timestamp). A total order could be established among these events only resorting on additional deterministic information such as the identifier of the publisher of the event, contradicting thus the anonymity principle. Moreover, such protocols can loose liveness during periods in which there is a disconnection with the NTP server.

3.7 Future work

The algorithm proposed in this Chapter solves the problem of total ordering in publish/subscribe middleware, but it is susceptible to further improvements. An aspect that we planned to address in the close future is the failure of topic managers. Currently, we assume the presence of reliable TMs; as such, they are always up and running during the timestamp generation of our protocol. On the contrary, we need to evaluate the impact of a failure of a topic manager on the correctness and execution of the algorithm, and how this affects the system performance. Then, we propose to study a solution that replicates the state of a topic manager on other nodes, in order to improve the reliability of our algorithm.

Another aspect that we plan to study is the impact of dynamic on the ordering algorithm. In particular, we want to evaluate how the change of subscriptions at run-time can impact both the timestamp generation and the dynamic topic ordering procedure. In this Chapter, we described how to manage subscriptions and unsubscriptions in order to preserve the Total Notification Order property, but how they can affect the execution of the algorithm and the topic adaptation to the publication popularity is still unclear.

Finally, a further improvement of our algorithm is to introduce causality

relation not only among events published on the same topic, as shown in Section 3.3, but also among events published on different topics. We are currently planning to work on an extension of our logical timestamping mechanism by using additional physical clock information that allow to notify events in an order that is coherent with the publication time, while both keeping the liveness of the notifications and the anonymity of the publisher/subscriber paradigm.

Chapter 4

Total order application to active database in cloud computing

In Section 1.2.3 we described the case study of active database in cloud computing, in which we argumented how an ACID-based transactional mechanism is not suitable for guaranteeing data consistency of replicated databases, because this would introduce an unsubstainable load of interactions and synchronizations among cloud nodes that would hamper the scalability of the system [29]. As such, the major cloud providers are moving toward *eventual consistency* [109] mechanisms in which cloud nodes are maintained in transient divergent states, from which they will eventually converge to a consistent one. To this end, all operations performed on database replicas must be *serialized*, i.e., a total order among operations must be defined to provide to each cloud node the same ordered sequence of operations.

The simplest solution to implement eventual consistency is to use an algorithm that delivers events as soon as they arrive and a rollback technique that corrects a *wrong* order. However, it has many drawbacks: first of all, in a large scale system with multiple event producers this simple solution does not help to infer which order can be defined as *correct*, and, then, when an event can be safely notified. It would require a consensus-based algorithm on the receivers' side, that would not solve the problem of strong coordination and synchronization among nodes. As suggested in [29], even if consensus algorithms like Paxos [77] are currently used in cloud computing (as an example,

in Google AppEngine), application developers are under huge pressure to use them only if strictly necessary.

The second problem that arises from the trivial solution is the high number of rollback operations that may be required to achieve a consistent event order among all database replicas. In fact, if an event is notified out of order, all events that have been processed so far must be undone and processed again in the correct order [73]. If we assume that an event represents a database transaction, this means that a transaction cannot be committed until the correct order among events has been achieved. This obviously translates into a delay penalty. In addition, rollback techniques pose issues in terms of number of stable storage accesses and interference with live processes: in fact, typical recovery protocols [69, 70, 100, 102] require either the entire system blocks, or live processes refraining from receiving messages during recovery [48]. As such, a solution for eventual consistency in cloud computing should ensure a total order among events that avoids, or, at least, reduces the number of rollback operations.

In the remainder of this Chapter we show how the total order algorithm described in Section 3.2.2 can be applied to the context of active database in cloud computing to ensure eventual consistency among all replicas. We compare our solution with a gossip-based protocol described in [16] in terms of *end to end latency* (as defined in Section 3.5.1), and *percentage of rollbacks*, i.e., the fraction of transactions that are rolled-back before their commit due to a inconsistent event notification order.

4.1 System model and problem statement

Figure 4.1 illustrates the application scenario: a database system is composed by several data centers connected through a wide area network. Each database within a data center, also referred to as cloud node, stores a portion of the whole information. In addition, we assume that the same piece of information is replicated on a set $x_1, x_2, ..., x_n$ of n replicas geographically distributed across the global system.

Users and data management systems, generically referred to as clients, can access the database system to execute *read* and/or *write* operations: a *read* corresponds to a query to retrive an information, while a *write* consists in updating data stored in the system. When a client executes an operation, this operation is routed toward one of the replica that contains the required information. In addition, in case of a *write*, the replica that performs the operation also updates all other replicas.

With reference to the eBay architecture described in Section 1.2.3, we can think about the global system as one of the database systems in which

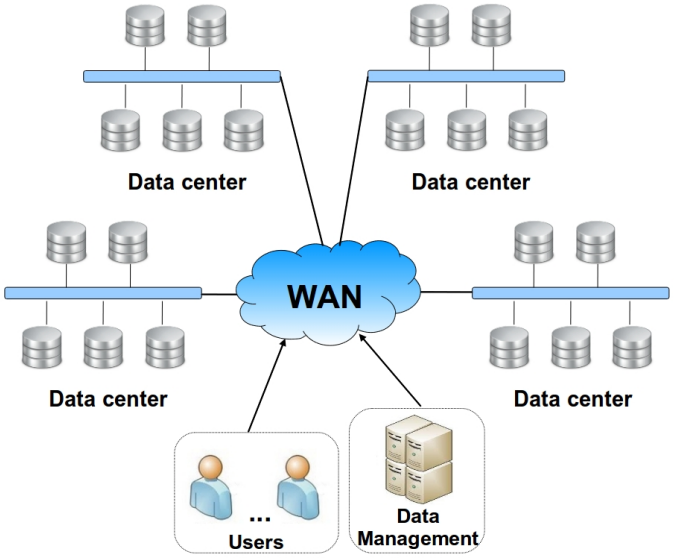


Figure 4.1: Example of a database system composed by several data centers connected through a WAN.

the architecture is partitioned: Users, Item, Transaction, Product, Account, Feedback. Sets of replicas store part of the whole information: as an example, if we consider the Product database system, a set of replicas stores information about sport, wear and gadgets, another set stores information about hi-tech, electronic devices and software, and so on...

The communication within the database system takes the form of events exchange, where an event represents an operation that has to be executed on the replica that receives it. These operations are specifically defined as *transactions*, i.e., sequences of *read* and/or *write* operations followed by a *commit* or *abort*. The correctness criterion for transactions that we consider is *linearizability* [64], also known as *one copy serializability* [25]: an object is perceived as one logical copy despite the existence of multiple copies, and the system allows only serializable transactions. To ensure linearizability, we have that the event delivery must satisfy the following two properties [62]:

1. *Order*: if two replicas x_i and x_j notify two events e and e', they notify them in the same order.

2. *Atomicity*: if a replica notifies an event e, then every other correct (non crashed) replica also notifies e.

The problem that we want to address is to guarantee eventual consistency among sets of replicas by reducing the number of rollbacks required to provide the same ordered sequence of operations to all cloud nodes within a set. A high number of rollbacks, in fact, may have a strong impact both on the performance and the correctness of the whole system. As also stated in the previous Section, typical rollback algorithms require either the system stops or that live processes refrain from receiving new events. Even if scalability in cloud computing can be achieved at the cost of a higher latency [29], blocking continuously the system due to recovery operations may dramatically enlarge the delay between a user's request and a system's response. On the other hand, refraining a running process from receiving new events may lead to discard some of them, expecially in presence of a high event rate: this, in turn, may enlarge the inconsistency window [109] (see also Section 1.2.3), so to produce unpredictable results to users' queries until all missed events have been recovered.

4.2 Architectural aspects

The architecture that we use to address the problem of guaranteeing eventual consistency among database replicas within a cloud by reducing the number of rollback operations is depicted in Figure 4.2. In particular, it consists of the architecture described so far in this thesis, with an ordering layer on top of a topic-based ENS, with the addition of a module that guarantees the atomicity property for each notified event. This module defines how a transaction terminates; we assume that one of the two mechanisms described in [112] is used to commit a transaction:

- voting: requires a message exchange to coordinate replicas or a confirmation message sent by a given site;
- non voting: replicas can *deterministically* decide on their own whether to commit or abort a transaction.

How the atomicity module is implemented in practice is out of the scope of this work.

In addition, we consider all replicas being both publishers and subscribers of events. External clients, instead, act as publishers that can execute operations on any replica. In case of a *write*, a replica updates its information and publishes it as a new event. At each subscriber's site, a new event is notified by the cloud node respecting order and atomicity properties.

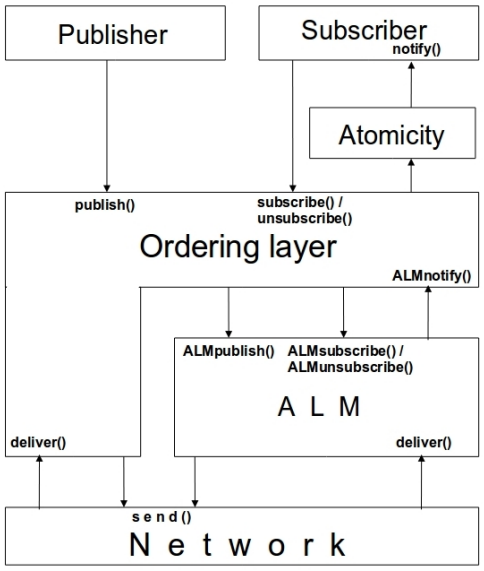


Figure 4.2: Architecture implemented by all cloud nodes.

In the experimental evaluation we compare our algorithm to the one described in [16]. The reason why we have chosen this solution for a comparison lies in the fact that it addresses the same problem that we have defined above. Specifically, the algorithm in [16] ensures *unconscious* eventual consistency. The term *unconscious* is used to evidence that, despite all system nodes eventually will see the same ordered sequence of events, in practice they do not know when all of them are actually consistent. To ensure ordering among events, the algorithm in [16] uses a *coalition* of k sequencers, one of which is randomly selected to assign a sequence number to a new event before it is sent to all other nodes by means of a gossip protocol. A coalition member assigns to an event e a sequence number of the form $sn_e = n|c| + pid$, where n is a monotonically increasing variable, $|c|$ is the cardinality of the coalition and pid is the node identifier. This function ensures that no two events have the same sequence number, but not that the sequence numbers provided by different sequencers are consecutive. A total order is defined among all events by using the sequence number and the sequencer identifier: for any pair of events e and e', e precedes e' if and only if (i) $sn_e < sn_{e'}$ or (ii) $sn_e = sn_{e'}$ and $pid_e < pid_{e'}$.

The first time that a system node wants to send an event, it has to discover an existing coalition. If it does not find one, at the expiration of a timeout it creates a new coalition. To do that, the node includes itself and some other nodes in the new coalition to get the desired size. However, the target of the

algorithm is to exploit the time intervals in which the network is *"synchronous enough"* to merge all coalitions in a single one of size k. Finally, due to the fact that sequence numbers provided by different sequencers may be not consecutive, a rollback technique is used to correct the wrong order.

In both solutions, we assume the presence of a standard recovery algorithm like [25] to execute a rollback operation.

4.3 Experimental evaluation

4.3.1 Setting

We consider the database system depicted in Figure 4.1 as composed by two geographically distributed data centers of five replicas each. A replica is a virtual machine with 1 GB RAM and running Ubuntu 10.4 OS. It represents a cloud node that has installed the prototype of our ordering algorithm deployed on top of the SCRIBE publish/subscribe middleware, or the algorithm presented in [16]. In the latter case, we consider a single coalition of size k. Nodes within a data center are connected through a high speed local network [2], while the WANem emulator is used to emulate the presence of a WAN between the two data centers, with mean link delay set to 21 milliseconds.

We consider the information stored in the database system as divided in categories, each one representing a different topic. As an example, if a set of replicas stores information about sport, wear and gadget categories, we consider those cloud nodes as producers and consumers of events of three different topics. Despite typical database systems are divided in several sets of replicas, for simplicity sake we consider a single set in which cloud nodes are producers and consumers of events of a number of topics varying in the set $\{1, 5, 10, 15, 20, 30\}$. In addition, we model the topic selection for a database *read* or *write* operation by a uniform distribution.

The metrics we evaluate in our experiments are the *end to end latency* and the *percentage of rollbacks*, as previously defined, while the parametres we vary are the event rate, in the range [1-100] events/sec, and the buffer size, in the range [0-100]. Finally, each reported value is the mean of at least three different experiments over the entire set of nodes, in runs of five minutes each.

4.3.2 Results

Figure 4.3 shows the percentage of rollbacks between our algorithm, referred to as TO, and the one presented in [16], referred to as UEC. In this experiment

we vary the event rate from 1 to 100 events/sec, while the number of topics is fixed to 5. For the UEC algorithm we consider the cases in which the coalition size k is 1, 2 or 3. The Figure shows that TO largely outperforms UEC with 2 and 3 sequencers, and for a lower event rate also the solution with a single sequencer.

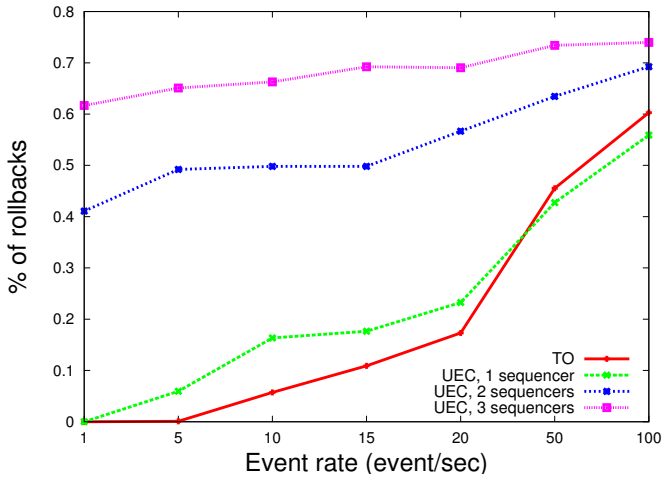


Figure 4.3: The collaborative timestamp generation process in TO largely reduces the percentage of rollbacks with respect to UEC.

The rationale behind this result lies in the way the two algorithms assign sequence numbers: while TO relies on a cooperative procedure that generates a non decreasing sequence of timestamps, as described in Section 3.2.2, the coalition members in UEC may generate a not consecutive sequence of values. As an example, consider the following scenario, with a coalition formed by three members, p_1, p_2 and p_3, respectively with process identifier 1, 2 and 3. Let us consider three consecutive sequence numbers assigned by p_2: according to the formula $sn = n|c| + pid$, p_2 generates the sequence 2, 5, 8. Now let us further consider two sequence numbers generated by p_3 and p_1: they respectively assign 3 and 1. Because the correct (total) order is 1, 2, 3, 5, 8, the delivery of events with sequence numbers assigned by p_3 and p_1 require two rollback operations. This phenomenon is further exacerbated by a high event rate and the presence of a WAN, that may also prevent an ordered delivery of events timestamped by the same sequencer.

Augmenting the event rate, Figure 4.3 also shows that the percentage of rollbacks for TO is higher than the one for UEC with one sequencer. However, the rationale behind this result is shown in Figure 4.4: the UEC algorithm (in presence of 1, 2 and 3 sequencers) discards events when the event rate reaches

50 or 100 events/sec. This behavior is motivated by the gossip algorithm used for event dissemination: due to the probabilistic nature of gossip-based protocol, we noticed that an event in UEC needs to be sent almost five times more than in TO, in order to reach all destination sites. This obvioulsy generates an overhead that leads cloud nodes to discard a fraction of events when the event rate increases.

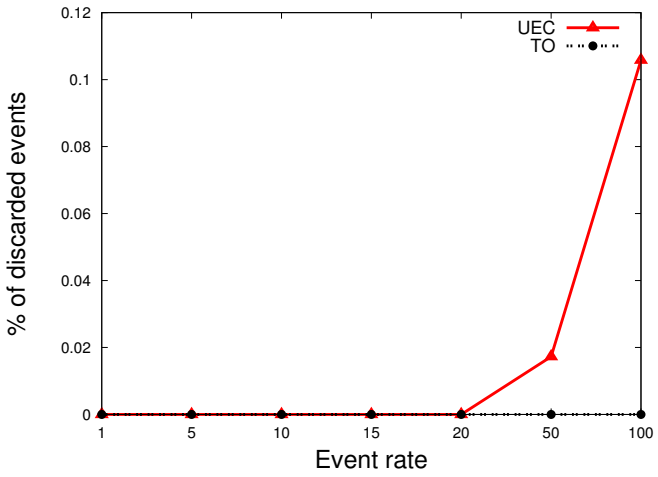


Figure 4.4: Augmenting the event rate, UEC discards events due to the usage of a gossip based protocol.

The reduction of the percentage of rollbacks obtained with TO comes at the cost of a higher latency by increasing the number of topics from 1 to 30. Figure 4.5 shows that, while UEC delivers always the same performance, the latency for TO increases almost linearly with the number of topics. This is due to the timestamp generation procedure, that involves an increasing number of topic managers.

Finally, we also evaluated the trade-off between the percentage of rollbacks and the latency for the TO algorithm. We fixed the event rate to 50 events/sec, the number of topics to 5 and the buffer timeout for each event to 1 second. Figure 4.6 shows how the need for a recovery procedure can be further mitigated by increasing the buffer size, at the cost of a higher latency. On the contrary, a buffering technique is meaningless for the UEC protocol, as also stated in [16], because each event brings no information about other possible events in transit[1].

[1] A buffering technique is used in presence of several coalitions. In the scenario simulated above, in presence of a single coalition, the usage of a buffer brings no benefit to the algorithm.

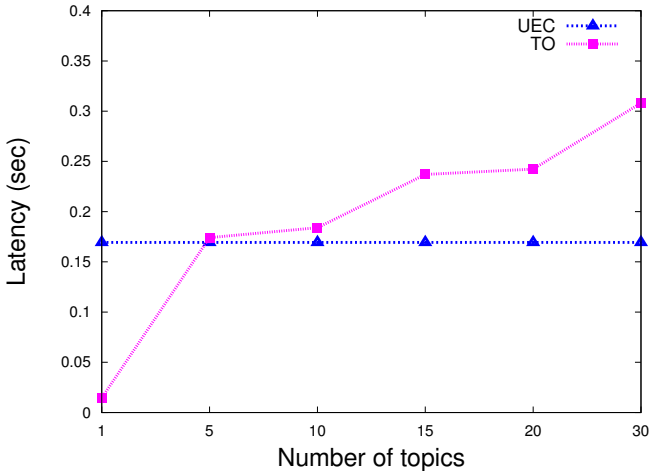


Figure 4.5: Augmenting the number of topics UEC does not decrease the latency performance, while TO has an almost linear increasing behavior.

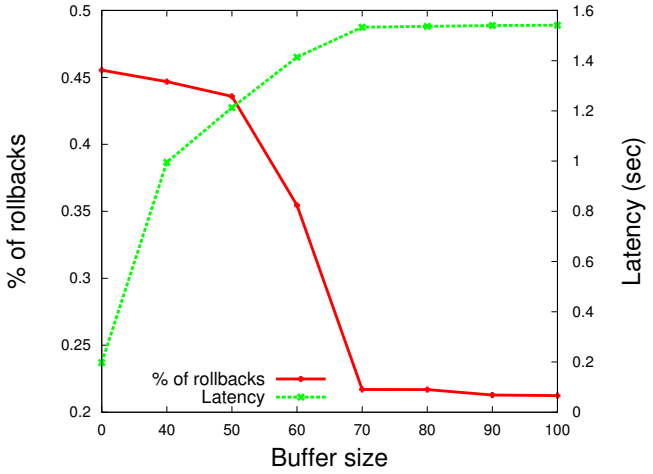


Figure 4.6: A buffering technique can reduce the percentage of rollbacks with TO at the cost of a higher notification latency.

The little improvement that we have when moving from buffer size 0 to 40 is due to the fact that the buffer is too small to accomodate events published at a rate of 50 per second. As such, the buffer is always full, and the event at the head of the buffer, say e, has to be delivered as soon as a new one has to be enqueued, even if the notification of e should wait the notification of prior

events. In addition, Figure 4.6 shows another interesting result: after a given size, i.e., 70 events, there is no benefit of using a larger buffer. This is motivated by the way in which the TO algorithm builds timestamps. In particular, the following scenario may occur: an application publishes n events on the most popular topic, say T. Later, the application publishes two different events e' and e respectively on the topic following T in the popularity order, say T', and on T. The events e' and e have a timestamp $\{< T, n >, < T', 1 >\}$ and $\{< T, n + 1 >\}$ respectively. A subscriber that receives both e' and e cannot infer which of the two should be notified first. This is due to the fact that the timestamp of e' includes also a sequence number for events published on topic T, but the timestamp of e does not include information about publication on T'. The choice we did in the design of the TO algorithm was to reduce the overhead due to the use of timestamps, by modifying at run-time the topic order to reflect the publication popularity. In the context of active database in cloud computing, this choice imposes a trade-off between overhead and percentage of rollbacks, that, however, is limited to a mean value of 21% for a medium/high event rate.

4.4 Future work

In this Chapter we have shown how our total order algorithm can be used to ensure eventual consistency in a database cloud reducing the need for rollback operations. In the close future we plan to extend our evaluation to a more complex scenario, in which the database system is composed by more than two data centers, each one with several sets of replicas (i.e., cloud nodes are subscribed to different topics). In addition, we think to extend this evaluation by comparing the TO algorithm with a solution based on a group communication toolkit like JGroups. In this case, an interesting aspect would be to evaluate the trade-off between notification latency and number of events delivered to subscribers that do not belong to their subscriptions. A group communication toolkit, in fact, is known to provide a fast event diffusion within a single group of processes, even if we have already shown that the performance may degrade in WAN (see Figure 3.24 in Section 3.5.2). As such, due to the presence of several sets of replicas, cloud nodes will also receive events to which they are not subscribed. On the contrary, in the TO algorithm each node receives only events published on subscribed topics, which largely reduce the usage of network resources. On the other hand, the time spent to generate a timestamp is affected by the number of topics, due to a cooperation among topic managers.

Finally, another aspect that we plan to investigate is how the throughput varies augmenting the event rate and the number of topics in the system.

Chapter 5

Timeliness and reliability issues

Information over wide area communication channels may be affected by link failures and network device congestions, which can determine burst of message losses [85] or delays. A similar behavior can compromise the execution of a critical infrastructure: as such, the publish/subscribe middleware should strive to enforce a *reliable* and *timely* event notification to all intended destinations. As an example, in the ATC case study presented in Section 1.2.4 we described the flight plans exchange scenario between several flight processors. We stated that this data contains information about the trajectory and the coordinates of an aircraft *en route*; if such an event is lost or delayed, a flight processor may not be able to infer the current position of the aircraft and to detect possible collisions.

The research community on publish/subscribe services has investigated proper methods to implement a reliable event dissemination, and several approaches, such as [27, 71, 92] have been already proposed in the last decade. Most of the current solutions for a reliable event dissemination only focus either on the use of the TCP transport protocol, or on retransmission techniques, and any reliability improvement is always gained at the expenses of severe performance overhead or fluctuations. Therefore, these approaches are not suitable for modern SoS where timeliness matters as well as fault-tolerance.

In this Chapter we aim to fill this gap by proposing a strategy to achieve both reliability and timeliness in event dissemination performed by publish/subscribe services over WAN. To this end, we combine coding and gossiping so to achieve the best from both: coding is known to reduce the notification deliv-

ery time [78], while gossiping is known to improve the reliability. We consider a publish/subscribe middleware deployed over Internet links that exhibit a non negligible probability to have bursty losses. Subscribers apply a gossip strategy to recover from possible lost events, with the possibility to exchange coded information, too. In the remainder of the Chapter, we provide a theoretical model to evaluate the potential benefit of coding on the information delivery performance. These results are also confirmed by an experimental analysis conducted on a real air traffic control workload, which evidences how coding mitigates latency and overhead penalties to ensure reliable event notification.

5.1 System model

We consider each system node implementing the architecture shown in Figure 5.1, where the *Timeliness and Reliability* layer is deployed on top of a generic topic-based ENS.

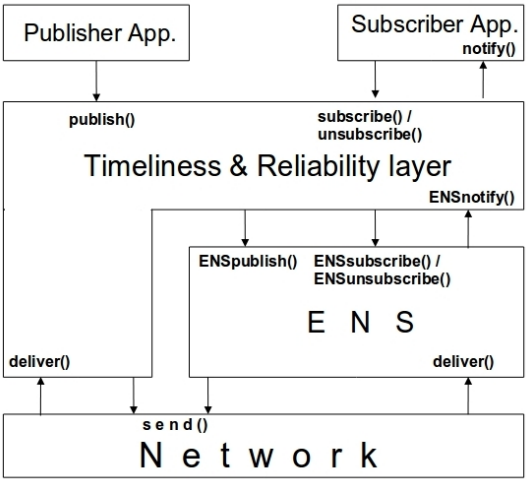


Figure 5.1: Architectural view that shows how the timeliness and reliability module acts as a mediating software layer between the applications and an existing event notification service.

We assume that links among nodes are not reliable, and exhibit a loss pattern characterized by *Packet Loss Rate* (PLR), which is the probability to lose a packet, and *Average Burst Length* (ABL), which is the mean number of consecutive lost packets. In particular, the adopted network model is the *Gilbert-Elliott* [47, 59], one of the most-commonly applied in performance

evaluation studies, due to its analytical simplicity and the good results it provides in practical applications on wired IP networks [75, 116]. As depicted in Figure 5.2, the Gilbert-Elliott model is a first order Markov chain with two states: "Good", with a state-dependent error rate equal to $1 - K$, and "Bad", with a state-dependent error rate equal to $1 - H$. Typically, K is assumed equal to 1 so that the "Good" state implies that no losses are applied, while in our model we assume that $1 - H$ is equal to PLR.

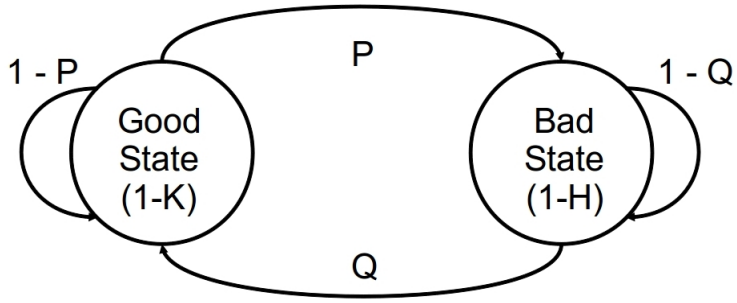


Figure 5.2: The Gilbert-Elliott model as a 2-state Markov process.

The adopted network model is characterized by four transition probabilities:

- the probability P to pass from state "Good" to state "Bad";
- the probability $1 - P$ to remain in state "Good";
- the probability Q to pass from state "Bad" to state "Good";
- the probability $1 - Q$ to remain in state "Bad".

Given PLR and ABL, it is possible to compute P and Q as follows [63]:

$$P = \frac{PLR \cdot Q}{1 - PLR} \qquad Q = ABL^{-1}$$

In addition, $K = 1$, means that no packets are lost when the model is in the "Good" state, while $H = 0$, means that all packets are lost when the model is in the "Bad" state. It is simple to demonstrate that, given such a model, the probability to lose a packet is equal to the mean PLR over time. Finally, the assumed fault load consists in omissions only, that have been applied at the overlay level. This is motivated by the consideration that it is possible to have a simple estimation of losses due to the Internet by means of application-level measurements, while network-level measurements are difficult, or even impossible to obtain.

5.2 Introduction to network coding and gossiping

Linear network coding. Linear network coding [56] is a technique that allows to convey the information content of n original packets, $x_1, x_2, ...x_n$, as a set n linearly independent combinations (encoded packets) and to easily generate redundancy virtually for any lost packets by means of linear combinations of original data. Each linear combination is given by

$$y_i = \sum_{i=1}^{n} c_i x_i$$

where coefficients c_i are taken uniformly at random over the set $0, ..., q-1$, with the all zero coefficients case excluded. All operations are performed over the Galois Field $GF(2^w)$, with $2^w = q$. Each coded packet y_i is equipped with the coefficients c_i used to produce that packet and that will be used by destinations in the decoding phase. Note that the overhead due to coefficients is very modest, as it is equal to send nw additional bits (for example, for n = 10 and q = 8 this means just 30 bits; considering a typical packet size of 1KB, the overhead is less than 0.4%).

Linear network coding increases the capacity of a network for multicast flows [56, 78]. Consider, as an example, a network with M sources and N destinations [56]. By assuming that, without network coding the source rates are such that the network can support each destination in isolation (as it was the only destination in the network), in [78] the authors show that, in presence of coding and with an appropriate choice of linear coding coefficients, the network can support all destinations simultaneously. As such, when the N destinations share the same network resources (physical or overlay links), each of them can receive information at the maximum rate, as it was using those resources by itself. This clearly improves the throughput and reduces the event distribution latency.

To get a better insight of the benefit of coding, let us compare a protocol that uses coding against a plain protocol. The power of network coding with respect to a plain data dissemination is particularly evident when a node introduces a redundancy. Let us consider two different cases:

- a node sends n plain packets plus a encoded packets, with $a < n$, generated as previously described;
- a node sends $n + a$ plain packets, with $a < n$.

Now let us consider a receiver that receives just n' out of the n packets (i.e., $n - n'$ packets have been dropped by the network); in addition, it receives all redundant packets, such that the total number of received packets is $n' + a \geq n$.

In case of coding, if coefficients c_i are independent, then any of the a additional packets can be useful to fully reconstruct the information. On the contrary, without coding, the a packets could be just a copy of the n' packets; in this case, the redundancy is not useful to reconstruct the whole information. This characteristic of network coding also allows a node to generate redundancy (i.e., linear combinations) even if it has received just a partial information.

Gossip-based solutions. The gossip paradigm [74] is based on the so-called *epidemic approach*, where an event is disseminated like the spread of a contagious disease or the diffusion of a rumor. Specifically, in a gossip-based protocol, a node stores a received message in a buffer of size b, and forwards it a limited number of times t to a randomly-selected set of nodes of size f. Many variants of gossip algorithms exist [46]:

1. *Push Approaches*: messages are forwarded to the other nodes as soon as they are received;
2. *Pull Approaches*: nodes periodically send to other nodes a set of recently-received message identifiers; if a missing message is detected by comparing the received set with the local history, then a transmission is requested;
3. *Push/Pull Approach*: a node forwards to the other nodes only the identifier of the last received message; if one of the receivers does not have such a message, then it makes an explicit pull request.

Gossip algorithms are characterized by two parameters:

1. *Fanout, f*: the number f of contacted nodes by a gossiper during a single gossip round.
2. *Fanin, t*: each node has an history of the identifiers of received events that is accessed when a pull or push/pull round starts. The fanin indicates the number of rounds t that an identifier is sent to gossip partners, after which it is deleted from the history.

Gossip-based protocols have several advantages that have been thoroughly studied: few initial infection points are sufficient to quickly infect the whole population as the number of infected processes grows with an exponential trend. Moreover, these algorithms are also strongly resilient to the premature departure of several processes, making them very robust against failures. The gossip approach has been successfully applied to a variety of application domains such as database replication [46], cooperative attack detection [117], resource monitoring [107], and publish/subscribe based data dissemination [41]. Taking into account the properties of an ideal event dissemination

service, most of such algorithms based on the gossip paradigm are able to deliver a huge amount of events in a geographically distributed setting with nice reliability properties. Thanks to the quick spread of *infection* also the time figures are very interesting.

5.3 Achieving reliability and timeliness in WAN

5.3.1 The protocol

We consider the *Timeliness and Reliability* layer as a composition of two independent blocks: (i) one implements a network coding protocol that aims to reduce the delivery time of an event over the entire set of subscribers, as motivated in the previous Section, and (ii) one implements a gossip-based algorithm to recover from possible event losses. To this end, we assume that each node can access a peer sampling primitive [66, 86] to obtain a sample (i.e., another subscriber interested in the same topic) to gossip with.

The protocol we use to reliably and timely deliver events to subscribers is composed by two phases:

1. *Dissemination:* the publisher divides an event into n plain packets and publishes them over the ENS. Moreover, a additional packets are also published. We call these the *redundancy* of the protocol. We separately consider two different cases: (i) *no-coding:* the a packets are randomly selected among the n plain packets; (ii) *coding:* the a packets are linear combinations of the n packets;

2. *Recovery:* subscribers use a gossip protocol to gather possible lost events. This phase strictly depends on the gossip style in use: if the push or the push/pull strategy is enabled, then, a subscriber that notifies an event (i.e., it has received enough packets to reconstruct the whole event) disseminates to the other f subscribers the received packets or the event identifier respectively. When the pull style is enabled, periodically a subscriber disseminates to other f nodes the identifiers of the last notified events.

A consideration needs to be done: the delay introduced by encoding and decoding operations is low for two reasons: (i) on the encoding side, the operations take not so much time because they are simple linear combinations and several libraries can be used to perform them efficiently; (ii) on the decoding side, operations are progressive, i.e., they are based on a *decoding matrix* that is built concurrently with the reception of packets. The decoding matrix is maintained in the triangular form by using Gaussian elimination [56]. Each

time a new encoded packet arrives, it is inserted into the matrix only if the new carried coefficients increase the rank of that matrix. Therefore, the decoding delay overlaps with the transmission delay. Further details about the delay penalty introduced by the coding operations can be found in [56].

In the following, before describing the algorithm in detail, we introduce the local data structure maintained by publishers and subscribers.

Local data structure to each publisher p_i: each publisher maintains locally the following data structures:

- id_e: is a unique identifier associated to an event e produced by p_i.
- *packets*: is a set variable, initially empty, that contains all packets in which an event is fragmented.
- *redundancy*: is a set variable, initially empty, that contains redundant packets: they can be plain or coded packets.
- *coding*: is a variable that indicates if coding is enabled.
- *publishedEvents*: is a set variable, initially empty, that contains the triple $\{e, redundancy, T\}$, where T is the topic on which that event has been published.

Local data structure to each subscriber s_i: each subscriber maintains locally the following data structures:

- *coding*: is a variable that indicates if coding is enabled.
- $incomingPackets_{id_e}$: is a set variable, initially empty, that contains all packets received for the event with identifier id_e.
- $notifiedEvents$: is a set variable, initially empty, that contains the tuple $\{e, e_{id}, T, t\}$, where t represents the gossip $fan-in$.
- $lastNotifiedEvents$: is a set variable, initially empty, that contains the identifiers of the recently notified events and it is used during a pull-based gossip recovery procedure.
- *gossip_mode*: is a variable that indicates the gossip strategy used in the recovery phase of the algorithm.
- *contacts*: is a set variable that contains the identifiers of the nodes returned by the peer sampling service.

PUBLISH() **Operation.** The algorithm for a PUBLISH() operation is reported in Figure 5.3. To simplify the pseudocode, we defined the following basic functions:

- fragment(e): fragments an event e in n plain packets.
- encode($packets, a$): implements the coding operation by generating a linear combinations of the n plain packets contained in $packets$.
- selectPacket($packets, a$): randomly selects a plain packets among the n contained in $packets$.

The PUBLISH() operation works as follows: the event e is fragmented in n plain packets and stored in the $packets$ data structure (line 01). Then, a redundant packets are generated (lines 02-05): depending on the variable $coding$ in line 02, these packets can be linear combinations of the original n packets (line 03), or random packets selected among the original ones (line 04). The $n + a$ packets are published on the ENS (lines 06-11) and, then, stored in the $publishedEvents$ data structure (line 12).

```
operation PUBLISH(e, T):

(01)  packets ← FRAGMENT(e);
(02)  if (coding = TRUE)
(03)     then redundancy ←encode(packets, a);
(04)     else redundancy ←selectPacket(packets, a);
(05)  endif
(06)  for each (pkt ∈ packets);
(07)       ENSpublish (< pkt, FALSE, id_e >, T);
(08)  endfor
(09)  for each (red ∈ redundancy);
(10)       ENSpublish (< red, coding, id_e >, T);
(11)  endfor
(12)  publishedEvents ← publishedEvents ∪ {e, redundancy, T};
(13)  packets ← {};
```

Figure 5.3: The publish() protocol for a publisher p_i.

Note that each published packet is also provided with the identifier id_e and a *boolean* value that indicates if coding is enabled. The n original packets are always published without coding (07).

NOTIFY() **Operation.** The algorithm for a NOTIFY() operation is reported in Figures 5.4 and 5.5. To simplify the pseudocode, we defined the following basic functions:

- decode(pkt, id_e): it implements the decoding process by maintaining a triangular matrix for packets related to the event with identifier id_e. It returns a decoded packet.
- canReconstructEvent($incomingPackets_{id_e}$): it is a *boolean* function that checks if the received packets are enough to fully reconstruct the event with identifier id_e. If so, the function returns $TRUE$, otherwise $FALSE$.
- reconstructEvent($incomingPackets_{id_e}$): it actually reconstructs an event e with identifier id_e from the packets contained in the $incomingPackets_{id_e}$ data structure.
- getPeer(f, T): it provides access to the peer sampling service by returning f random nodes currently subscribed to the topic T. f represents the $fan-out$ of the gossip algorithm.
- pushEvent(e, id_e, T, s_j): this function starts a push-based gossip procedure by sending to a subscriber s_j the received event e.
- pushEventId (id_e, T, s_j): this function starts a push/pull-based gossip procedure by sending to a subscriber s_j the identifier id_e of the received event e.
- sendRecentHistory($lastNotifiedEvents, s_j$): this function starts a pull-based gossip procedure by sending to a subscriber s_j the identifiers of the last received events contained in the $lastNotifiedEvents$ data structure.

The NOTIFY() operation works as follows: upon receiving a packet, the handlePacket function is called (line 01). This function implements the core activity for each received packet; we decided to separate it from the NOTIFY() operation in order to reuse HANDLEPACKET also for processing the incoming retransmitted packets during the recovery phase of the protocol. Depending on the kind of received packet (plain or coded, line 02), it can be simply added to the set of the incoming packets (line 03) or it requires a decoding process first (line 04). At each received packet, the algorithm checks if there are enough packets to reconstruct the event (line 07). If so, the reconstructEvent function actually reconstructs that event (line 08). Recall from Section 5.2 that receiving a number of packets equal or higher to n is a necessary but not sufficient condition to fully reconstruct an event. In fact, without coding it is required to receive n *different* packets, while with coding n *independent* linear combinations are needed. When an event is fully reconstructed, the subscriber triggers

```
upon ENSNOTIFY(< pkt, coding, e_id >, T):

(01)  handlePacket(pkt, coding, e_id, T);

---------------------------------------------------------------

function HANDLEPACKET(pkt, coding, e_id, T):

(02)  if (coding = FALSE)
(03)    then incomingPackets_{id_e} ← incomingPackets_{id_e} ∪ {pkt};
(04)    else incomingPackets_{id_e} ← incomingPackets_{id_e} ∪ {decode(pkt, id_e)};
(05)  endif
(06)  if (|incomingPackets_{id_e}| ≥ n);
(07)    then if (canReconstructEvent(incomingPackets_{id_e}) = TRUE)
(08)          then e = reconstructEvent(incomingPackets_{id_e})
(09)                trigger notify (e, T);
(10)                notifiedEvents ← notifiedEvents ∪ {e, id_e, T, t};
(11)                if (gossip_mode = PUSH ∨ PUSHPULL)
(12)                  then contacts ← getPeer(f, T);
(13)                endif
(14)                if (gossip_mode = PUSH)
(15)                  then for each (s_j ∈ contacts)
(16)                            pushEvent(e, id_e, T, s_j);
(17)                       endfor
(18)                  else if (gossip_mode = PUSHPULL)
(19)                  then for each (s_j ∈ contacts);
(20)                            pushEventId (id_e, T, s_j);
(21)                       endfor
(22)                endif
(23)          endif
(24)    endif
```

Figure 5.4: The notify() protocol for a subscriber s_i.

the notify operation (line 09) and inserts it in the set of received events (line 10). Then, if the push or push/pull gossip strategy is enabled, the subscriber asks to the peer sampling service a set of f random nodes currently subscribed to topic T (lines 11-13), and sends them the received event (lines 14-17 for the push-based style) or its identifier (lines 18-21 for the push/pull-based style).

Figure 5.5 shows the pseudocode for a pull-based gossip strategy. Periodically a subscriber generates a set with the identifiers of the last received

```
upon TIMEOUT():

(01)  for each (< e, id_e, T, t > ∈ notifiedEvents)
(02)      lastNotifiedEvents = lastNotifiedEvents ∪ {id_e};
(03)      t = t − 1;
(04)      notifiedEvents ← notifiedEvents / {< e, id_e, T, t >};
(05)      if (t > 0)
(06)        then notifiedEvents ← notifiedEvents ∪ {< e, id_e, T, t >};
(07)      endif
(08)  endfor
(09)  contacts ← getPeer(f, T);
(10)  for each (s_j ∈ contacts)
(11)      sendRecentHistory(lastNotifiedEvents, s_j);
(12)  endfor
(13)  lastNotifiedEvents ← {};
```

Figure 5.5: The expiration of the timeout fires a new pull-based gossip execution.

events (line 02) and sends it to f random nodes by means of the sendRecentHistory function (lines 09-12). The number of times that an event identifier can be sent to other nodes is regulated by the parameter t, i.e., the $fan-in$ of the algorithm. Each time an identifier is inserted in the $lastNotifiedEvents$ data structure, its value of t is decreased by one (line 03); when $t = 0$, the tuple related to that identifier is no more updated in the $notifiedEvents$ data structure (lines 04-06).

Finally, it is worth mentioning that the implementations of the pushEvent, pushEventId and sendRecentHistory functions for the push-, push/pull- and pull-based recovery strategy respectively, can consider either the retransmission of plain packets, or the retransmission of linear combinations. In the experimental evaluation presented in Section 5.5 we compare the two cases in order to show the improvement in event dissemination over WAN when random linear coding is enabled.

5.3.2 Network coding: motivations

The reason why coding is expected to highly improve a gossip based recovery protocol comes from the following simple property. Consider the n-dimensional vector space V over a Galois Field with base q, namely $GF(q)$. The number

of elements in V is $q^n - 1$, i.e., it is given by the number of ways we can multiply a base of V with n coefficients, excluding the all zero case. For the same reason, a k-dimensional V's subspace V_k has $q^k - 1$ elements. Hence the ratio between the number of elements in V_k and the number of elements in V_n is $\frac{q^k-1}{q^n-1} \approx \frac{1}{q^{n-k}}$. This is also the probability that a random vector of V belongs to V_k.

Now, consider the case when a single event e is divided into n packets, which are transmitted to a set of destinations using different overlay channels. Suppose that, due to packet losses, a destination gets k out of the n packets, i.e., the destination has missed $m = n - k$ packets. For reconstructing the whole event the destination starts to retrieve the missed m packets through a simple gossip mechanism consisting in contacting another destination and pulling one random packet from it. If the packets are sent without coding and assuming the contacted destination experienced an independent packet loss pattern, the probability that the pulled packet is useful is clearly $\frac{m}{n} = 1 - \frac{k}{n}$. However, if the source node sends random linear combinations of the original packets, then, due to the aforementioned property, the probability that the pulled linear combination is independent from the already received ones is the probability that the pulled linear combinations does not belong to V_k, i.e., $\approx 1 - q^{-m}$.

Figure 5.6 shows the probability as a function of m for $n = 10, q = 8$ in the two cases (coding and no coding).

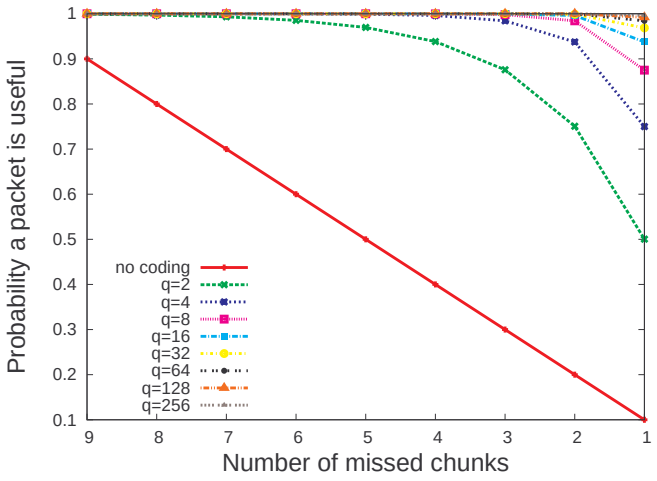


Figure 5.6: Probability that an additional random packet of data is useful as a function of the missed packets for $n = 10$.

While in the no coding case this probability increases linearly with the number of missed packets m, under a coding scheme the probability increases exponentially. In addition, for the coding case the probability to retrieve the last packet is as small as $1 - q^{-1}$ and then can be made very close to one by setting q to some convenient value, i.e., $q = 1024$. It is also worth noticing that, under our independence assumption, the problem of retrieving the m missed packets is the classical Coupon Collector's Problem [55], and, hence, the expected number of gossip rounds required to retrieve all the packets is $\Theta(m\, log(m))$, whereas using coding it is easy to see that the average number of rounds is $\Theta(m)$.

Finally, we remark that due to the linearity of the operations, encoded packets can also be obtained by linearly combining already encoded packets. This allows intermediate nodes in a multicast path to add redundancy exploiting the partial information content received so far thus reducing the end to end delay.

5.4 Protocol analysis under ideal conditions

In this Section, moving from the motivations discussed so far, we provide a theoretical analysis to show the benefit of using coding during the recovery phase of our algorithm with respect to a plain recovery protocol. In particular, we aim to show the reduction of the number of redundant packets and gossip interactions to fully reconstruct an event when network coding is enabled. Indeed, it can be considered as a means to also reduce the latency and the overhead for a reliable event delivery. For simplicity sake and without loosing of generality, several assumptions have been done:

- we consider a publish/subscribe middleware implemented as an overlay network that directly connects a single source to a set of receivers (i.e., there is no internal node between the publisher and subscribers);
- the publisher and subscribers' roles are played by the same node that constitute the ENS;
- the recovery strategy is based on a pull-style gossip strategy.

Some of these assumption will be relaxed in the experimental evaluation presented in Section 5.5.

The goal of the model is to capture the number N_R of gossip rounds required for a tagged destination node to retrieve an event composed by n packets. During each round, the destination performs F elementary operations, each called *contact*, consisting of contacting another node and pulling useful

data from it. The parameter F is also called fanout. We distinguish between two cases:

1. *coding*: the information retrieved to fully reconstruct an event consists in coded packets;

2. *no coding*: the information retrieved to fully reconstruct an event consists in plain packets.

We assume a discrete-time model, in which the contacts are numbered progressively. In the following we calculate $P_R(k) = Pr\{N_R \leq k\}$ for $F = 1$ and $F > 1$, i.e., the probability of retrieving the event in k rounds. We call this probability the *success rate*.

5.4.1 Success rate, $F = 1$

To numerically compute $P_R(k)$ we will exploit a Markov chain with $n + 1$ states and time variable transition probabilities. The state of the chain is the number of *useful* packets stored at the destination node at the end of the gossip rounds. For the no coding case, the useful packets correspond to packets with different identifiers, whereas for the coding case they are linearly independent combinations.

The state probability is denoted by $\pi^k(i)$, which represents the probability that after k gossip rounds the tagged destination stores i useful packets. We assume that the node contacted at k-th round did not execute yet the gossip operation at that round, i.e., the contacting destination node sees the contacted node as it was at the end of round $k - 1$. The evolution of the Markov chain is described through the following equation:

$$\pi^{(k)}(j) = \sum_{i=0}^{n} \pi^{(k-1)}(i) P_{ij}(k), \quad k > 0,$$

where $\pi^{(0)}(i)$ is the probability of storing i packets at the end of the dissemination phase of the protocol, and $P_{ij}(k)$ the state transition probability from state i to state j at gossip round k. This probability is $P_{ij}(k) = 0$ for $j < i$ (the number of stored packets cannot decrease) whereas $P_{nn}(k) = 1$. With these assumptions, the state n is an absorbing state and $\pi^k(n)$ represents the probability that at the end of round k the useful packets stored at the destination node is n, i.e., the event is fully received. Hence, $P_R(k) = \pi^k(n)$.

The transition probabilities as well as the initial state are computed in terms of the following three probabilities:

- *Holding probability* $P_H(k, r)$*:* probability of contacting a node holding r packets at the beginning of round k.

- *Increasing probability* $P_I(j|i,r)$*:* probability that after merging two random groups of i and r different[1] packets, the resulting group has j useful packets.
- *Loss probability* $P_L(n,d)$*:* probability of loosing d out of n transmitted packets over a Gilbert-Elliott channel, starting from a random instant of time.

As far as the transition probabilities are concerned, after performing a round the destination node increases its state from i to j if and only if the following events occur:

- The contacted node holds r packets, where $r \geq j-i+d$ and $d \geq 0$. This event occurs with probability $P_H(k,r)$.
- After adding these r packets the number of useful packets for the destination potentially becomes $j+d$, i.e., the number of additional useful packets is $j+d-i$. This happens with probability $P_I(j+d|i,r)$.
- The contacted node sends to the destination the $j+d-i$ useful packets and, during the transmissions, d packets are lost. This happens with probability $P_L(j+d-i,d)$

We then have:

$$P_{ij}(k) = \sum_{r=j-i}^{n} \sum_{d=0}^{r} P_H(k,r) P_I(j+d|i,r) P_L(i+d-j,d)$$

We now compute the key probabilities defined above as well as the initial distribution for the coding and no coding cases.

• *Holding Probability* - Our model assumes that the state of a node contacted during round k is equal to the state of that node at the end of round $k-1$. We also assume that the contacted node is itself experiencing the same sort of evolution as the contacting node, i.e., the *number* of packets stored by the contacted node follows the same statistic of the number of packets stored in the observed node, but delayed of one unit of "time"; hence

$$P_H(k,i) = \pi^{(k-1)}(i)$$

• *Increasing probability, baseline protocol* - The probability $P_I(j|i,r)$ for the baseline protocol can be found through a combinatorial argument. Let divide

[1] In the coding case *different* means linearly independent from each other.

the r packets into two subsets U of $a = j - i$ elements and $\overline{U}$ of $r - a$ elements, $0 \leq a \leq r$. This can be done into $\binom{r}{a}$ different ways. Now, let examine the packets according to some order (for example their identifiers). The first packet in U is useful with probability $\frac{n-i}{n}$, the second one with probability $\frac{n-i-1}{n-1}$, and so on. Hence, the elements in subset U are all useful with probability:

$$p_u = \frac{n-i}{n} \times \frac{n-i-1}{n-1} \times \ldots \times \frac{n-i-d+1}{n-d+1} = \frac{(n-a)!}{n!} \times \frac{(n-i)!}{(n-i-a)!}$$

Similarly, the first packet in the complementary subset $\overline{U}$ is not useful with probability $\frac{i}{n-d}$, the second one is not useful with probability $\frac{i-1}{n-d-1}$, and so on. Hence, all the $r - a$ elements of $\overline{U}$ are not useful with probability:

$$p_{\overline{u}} = \frac{i}{n-a} \times \frac{i-1}{n-a-1} \times \ldots \times \frac{i-(r-a-1)}{n-d-(r-a-1)} = \frac{i!}{(i-r+a)!} \times \frac{(n-r)!}{(n-a)!}$$

$P_I(j|i,r)$ is then the product of the above two probabilities times the number of possible subset of a elements:

$$P_I(j|i,r) = \binom{r}{a} p_u p_{\overline{u}}$$

• *Increasing probability, coding* - In Section 5.3.2, we have shown that the probability of a random linear combination being useful is bounded by $1 - \frac{1}{q}$, with the value q that can be chosen sufficiently high to make this probability almost one. Thus, we get:

$$P_I(j|i,r) = \begin{cases} 1 & j = i + r,\ i + r \leq n \\ 0 & \text{otherwise} \end{cases}$$

• *Loss probability* - This probability has been computed in several papers (as an example [113]); as such, we do not report further details. In Section 5.1 we assumed this probability equal to the PLR.

• *Initial distribution, baseline protocol* - Let assume that the source node sends the n original packets plus a additional ones, $0 \leq a \leq n$, in a random order, starting from a random instant of time. As the order of packet transmissions is random, the loss events are in turn randomized over the whole set of sent

packets. Hence, a loss hits any of the $n+a$ packets with the same probability. Now, there are

$$\binom{n}{d_1} \times \binom{a}{d_2}$$

ways to form a pair of groups, in which the first (second) group is a subset of d_1 (d_2) elements, taken from a set of n (a) elements. Therefore, the probability that n' out of the n original packet are lost and a' out of the a redundant packets are also lost, given that $d_1 + d_2 = n' + a'$ packets are lost, is

$$p(n', a') = \frac{\binom{n}{n'} \times \binom{a}{a'}}{\sum_{d_1=0..n}^{n} \sum_{d_2=n'+a'-d_1}^{a} \binom{n}{d_1} \times \binom{a}{d_2}}$$

The initial distribution can be found by summing up the probabilities associated to the following three events: (i) the total number of packet lost is $n' + a'$; (ii) among them n' are from the original n packets and a' from the redundancy; (iii) the number of different packets after merging the received $n - n'$ original packets and $a - a'$ redundant ones is i. Hence:

$$\pi^{(0)}(i) \quad = \quad \sum_{n'=0}^{n} \sum_{a'=0}^{a} P_L(n+a, n'+a') \times p(n', a') \times P_I(i|n-n', a-a')$$

• *Initial distribution, coding* - A random linear combination is very likely to be independent from any other group of random linear combinations. Hence, the initial distribution of the Markov chain is well approximated exploiting the received packet's distribution:

$$\pi^{(0)}(i) = \begin{cases} P_L(n+a, n+a-i) & i < n \\ \sum_{k=0}^{a} P_L(n+a, n+a-k) & i = n \end{cases}$$

5.4.2 Success rate, $F > 1$

In a gossip protocol with fanout F, during a round the destination node contacts F nodes and pull data from them. Although in real protocols these contacts occur in parallel, we treat them as elementary gossip operations occurring in sequence and changing the state of a node. For convenience, the initial state gets index $k = -1$, rounds are numbered starting from 0 and the first elementary operation of round 0 occurs at time index $k = 0$. Hence, for example, for $F = 2$, the contacts with number $k = 0, 1$ belong to round 0, contacts $2, 3$ to round 1, etc. In general, the $k-$th contact belongs to the round $\lfloor k/F \rfloor$. The node performing round F sees the contacted node being in the

state $F\lfloor k/F \rfloor - 1$. Under these considerations, the previous model can be still applied; the only formal changes are the definition of the holding probability

$$P_H(k,i) = \pi^{(F\lfloor k/F \rfloor - 1)}(i)$$

and the fact that the meaningful probability values are those for $k = jF - 1$, $j = 0, 1, 2, ...$

5.4.3 Results and discussion

In this Section we compare theoretical results with experimental results obtained through a custom time-driven simulator. We evaluate the *success rate*, i.e., the fraction of system nodes that fully recovers an event, where the system size is 100 nodes. This value is the mean of 100 different publications, averaged on all nodes. The network failure model is defined in terms of: (i) *Packet Loss Rate* (PLR), and (ii) *Average Burst Length* (ABL).

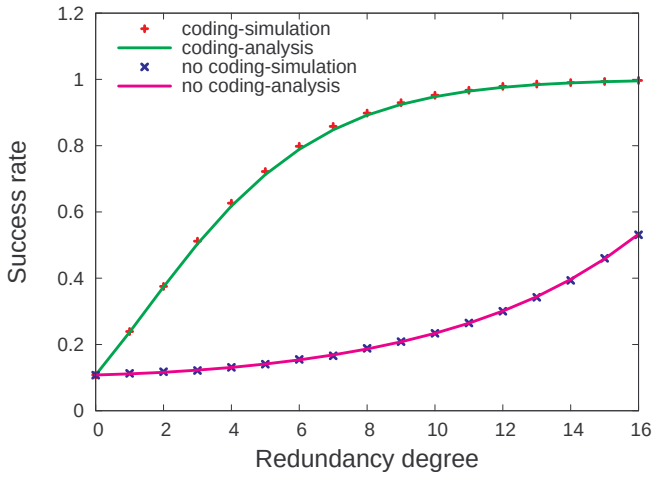


Figure 5.7: Reliability improvement in coding and no coding cases by varying the redundancy degree.

Figure 5.7 compares the success rate in coding and no coding cases, by varying the number of redundant packets. In both coding and no coding protocols, PLR is set to 0.2 and ABL to 2. We can see how coding exponentially improves the reliability by augmenting the number of redundant packets. This is direct consequence of the high probability for a linear combination to be independent from all the others. It is anticipated that a similar behavior is also found when a real publish/subscribe protocol is used.

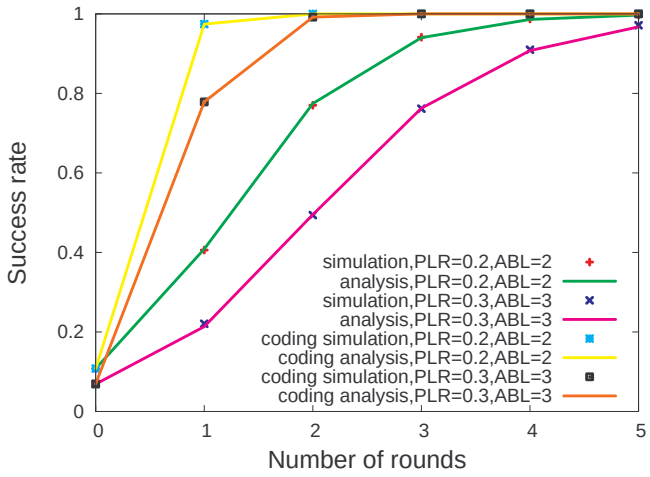


Figure 5.8: Impact of the number of gossip rounds on success rate.

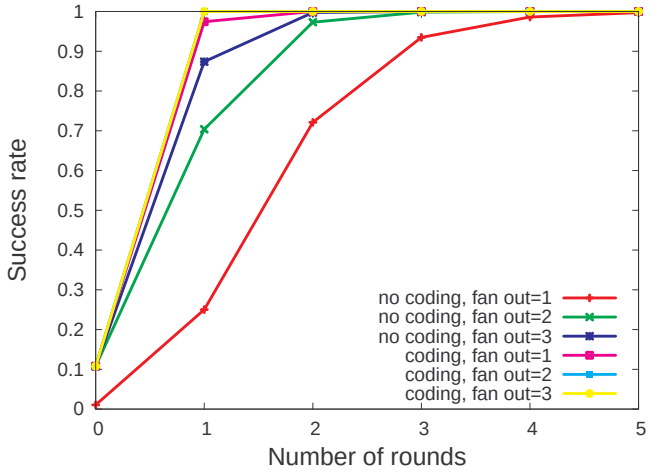


Figure 5.9: Effect of fanout on success rate in coding and no coding cases.

Figure 5.8, instead, shows how reliability augments in presence of gossip. In case of coding, 2 rounds are enough to fully recover the whole event. The Figure thus confirms that coding is expected to improve the delay performance as a consequence of a less number of rounds required to get the missed packets in case of loss. Finally, Figure 5.9 shows the effect of fanout for $PLR = 0.2$ and $ABL = 2$. By increasing the fanout, a node pulls a larger amount of data

per round, thus making a round more effective.

5.5 Experimental evaluation

The goal of this Section is to present experimental results that validate the theoretical analysis provided in the previous Section. We implemented our solution by using the OMNET++ [108] simulator: for the networking components we have used the INET framework, while as ENS we used the SCRIBE implementation provided by the OVERSIM [23] library.

The workload used in our experiments has been taken from the requirements of the SESAR project, since it is representative of a real LCCI case. Specifically, exchanged events have a size of 23KB, the event rate is fixed to 1 event/sec and the number of nodes is 40 (this is the estimated number of ATM entities involved in the first phase of the SESAR project, deployed in Italy, Switzerland and France). The network parameters are: link delay equals to 50 milliseconds, PLR = 0.05 and ABL = 2 [49]. We modeled the time to obtain a coded packet equal to 5ms, while the time for the dual operation is equal to 10ms. We also assumed the packet size equal to the payload of MTU in Ethernet (i.e., 1472 bytes) so that an event is fragmented in 16 packets.

Finally, without loss of generality, we considered a system with a publisher and 39 subscribers, all subscribed to the same topic. We simulated a period of 1000 publications and reported the average of three different experiments on the same scenario (we did not observe standard deviation above 5% of reported values, thus they are not plotted on the curves).

We evaluate the three different gossip strategies introduced in Section 5.2: *push*, *pull* and *push/pull*. The metrics we consider in our study are:

Success rate: the ratio between the number of received events and the number of the published ones, and it is referred to as reliability, which is the ability of the publish/subscribe service to deliver all the published events to all the interested subscribers. If success rate is 1, then all the published events have been correctly notified by all subscribers.
Overhead: the ratio between the total number of packets exchanged during an experiment and the number of packets generated by the publisher (that is the number of published events times the number of packets in which an event is fragmented to be conveyed by the network). It is a measure of the traffic load that the dissemination strategy imposes on the network, and should be kept as lower as possible, in order to avoid congestions.
Performance: mean latency is a measure of how fast the given dissemination algorithm is able to deliver events, while the standard deviation indicates pos-

sible performance fluctuations due to the applied fault-tolerance mechanisms, highlighting timing penalties that can compromise the timeliness requirement.
Dependance on Network Dynamics: mean latency in presence of different network conditions.

The parameters we vary in our analysis are:
Redundancy degree: number of redundant packets sent by the publisher in addition to the original ones.
Gossip fan-out.
Gossip fan-in: when not explicitly declared, we assume that the fan-in is set to 1.

In our evaluation, we report the results obtained to reach a success rate equal to 1 (thus, some curves may be truncated).

5.5.1 Success rate

In Figure 5.10 we compare the coding and no coding cases introduced in Section 5.4 by varying the redundancy degree.

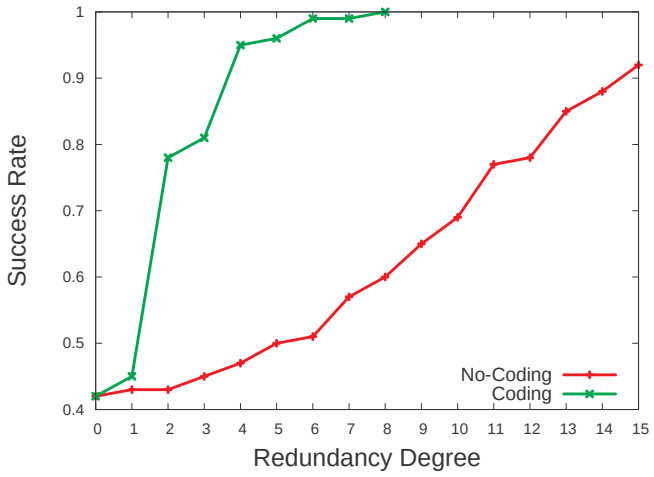


Figure 5.10: Success rate when transmitting plain or coded redundant packets.

The publisher publishes an event on SCRIBE, by sending several redundant packets. We recall that in the coding case, these packets are linear combinations of the original packets that compose the event. The obtained results confirm what we mentioned in the previous Section, i.e., coding is able to

improve the reliability of an event dissemination protocol without requiring a high redundancy degree. In fact, without coding a full success rate (complete delivery) is achieved only with a redundancy equal to 29 (i.e., the event is sent almost three times), while with coding this is obtained just with a redundancy of 8. This improvement is particularly meaningful for the considered ATC context, where real ATC systems currently ensure a complete delivery by sending a plain event three times[2].

In Figures 5.11 and 5.12 we evaluate the three gossip strategies, push, pull and push/pull, by varying the fan-out. The publisher periodically publishes an event on SCRIBE; in the push and push/pull cases, when a node completely reconstructs an event, it starts a gossip procedure by contacting a subset of the other nodes according to the fan-out value. On the contrary, in the pull procedure the set of the last received event identifiers is sent on a periodical basis, with the period fixed to 1.5 seconds.

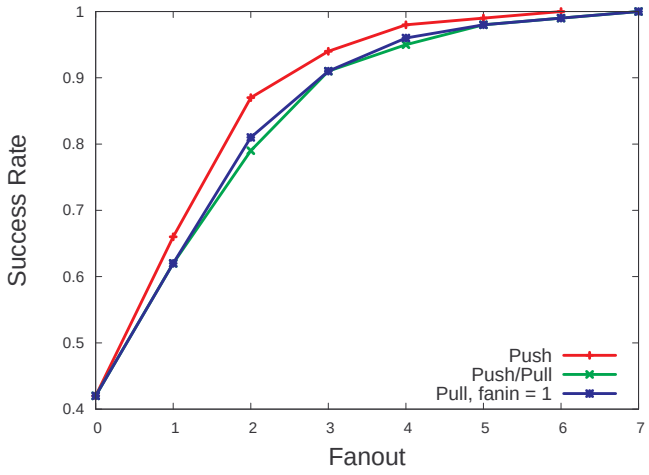


Figure 5.11: Success rate for the three gossip styles.

Figure 5.11 evidences that push is better than the other gossip styles because it forwards packets as soon as an event arrives; in pull and push/pull strategies, in fact, the loss of a message containing the set of the last received event identifiers or a retransmission request compromises the recovery of a given event. However, Figure 5.12 shows an interesting trade-off between fan-in and fan-out in the pull approach: by definition, augmenting the fan-in augments the number of times that an event identifier is sent to other nodes,

[2]Private discussion with Dr. Angelo Corsaro, CTO at PrismTech and co-chair of the Data Distribution Service Special Interest Group.

that in turn reduces the number of partners per round to succesfully spread this information.

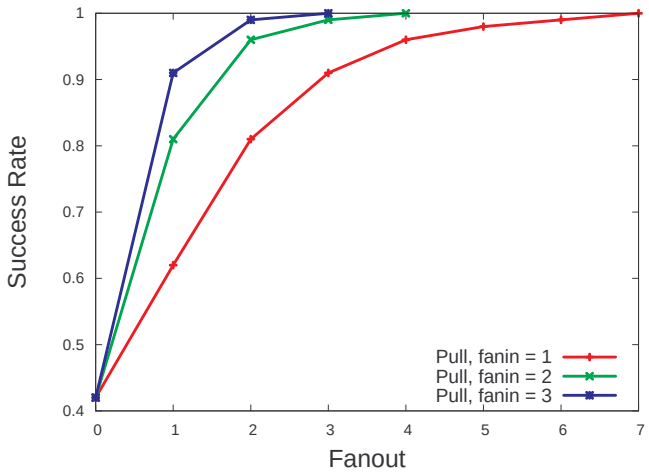


Figure 5.12: Trade-off between fan-in and fan-out to reach a full success rate in the pull-based gossip.

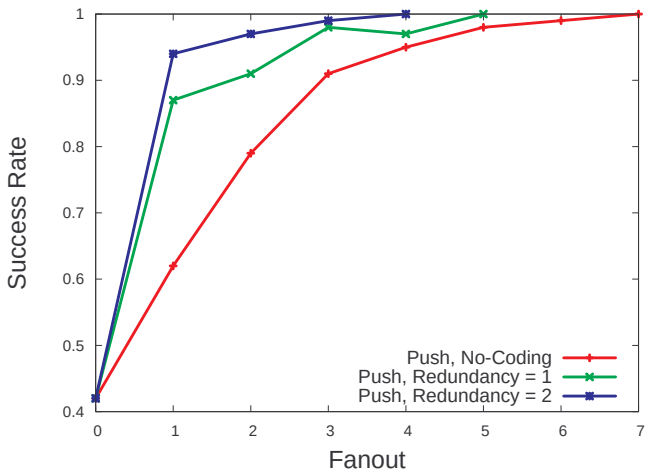


Figure 5.13: Fan-out to achieve a reliable delivery by varying the redundancy degree in the push-based gossip.

In Figures 5.13 and 5.14 we combine gossip and coding: the publisher sends a plain event, as before, and introduces a redundancy by sending one or two additional packets (i.e., linear combinations of that event). During the gossip phase, nodes retransmit just the redundancy. Results show that when gossip is teamed up with coding, a full success rate is obtained with a smaller fan-out, due to the ability of network coding to provide *useful* packets to fully reconstruct an event (under the assumption of independent coefficients) [56].

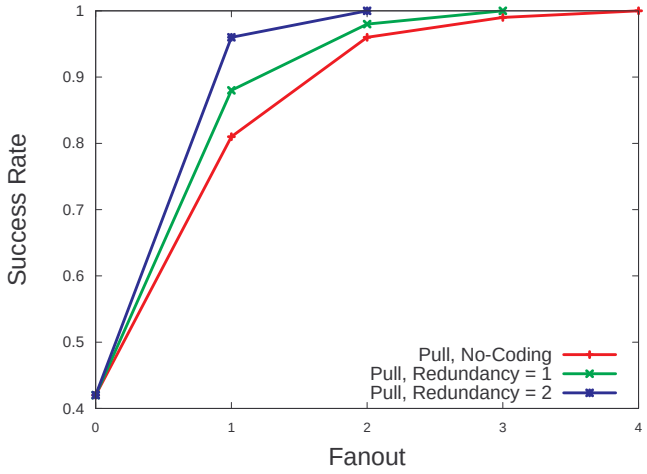


Figure 5.14: Fan-out to achieve a reliable delivery by varying the redundancy degree in the pull-based gossip.

5.5.2 Overhead

Figure 5.15 shows the overhead produced by the three gossip strategies, evidencing that it only depends on the fan-out, and not on the experienced network conditions (in fact, it does not change even varying ABL). A slight variation is just present in the pull-based strategy. This difference is due to its reactive nature: a retransmission request is issued only when a missing event is detected by looking at the sets of identifiers periodically received from other nodes. Augmenting the ABL augments the probability to loose a packet, and then to ask for a retransmission. On the contrary, in push and push/pull strategies, a gossip message is always sent as soon as an event is received. Moreover, the improvement introduced by push/pull with respect to the push style, by sending only the identifier of the received events, allows to reduce in a remarkable manner the experienced overhead. It is worth noticing

that the lowest overhead is in presence of pull gossip: as explained above, a retransmission is sent only when needed, so to reduce the overall traffic load.

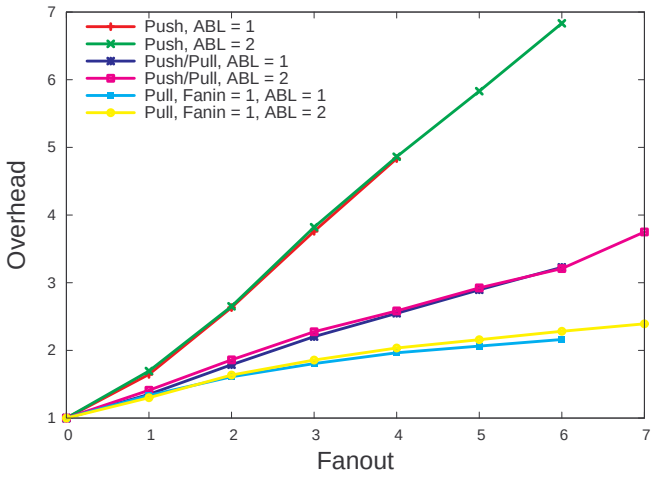


Figure 5.15: Overhead for the different gossip strategies by varying the fan-out. Each curve refers to a different network condition.

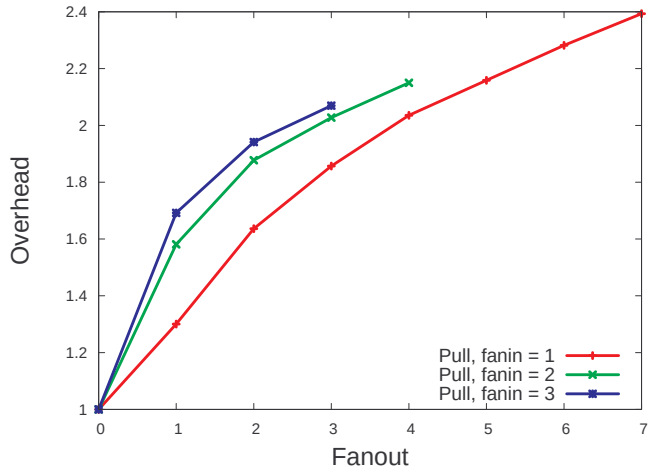


Figure 5.16: Overhead of pull gossip by varying the fan-out for different fan-in values.

In Figure 5.16 we separately analyze the impact of fan-in on the overhead

for the pull-based strategy. Due to the slight difference in the overhead value by varying the network conditions, for clarity of presentation we plot only the case with ABL = 2. As expected, augmenting the fan-in augments the probability that an event will be retransmitted several times, so as to increase the measured overhead. However, we have already seen that augmenting the fan-in reduces the fan-out to reliably notify an event.

Figures 5.17 and 5.18 show the impact of coding on push and pull gossip styles respectively.

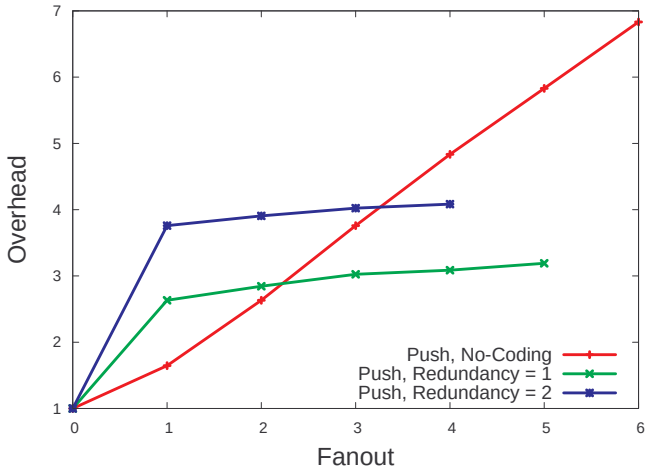


Figure 5.17: Overhead for combined push gossip and coding when redundancy degree is varied.

We notice that the redundancy initially introduces a higher overhead, but it is mitigated by the lower fan-out required to achieve a success rate equal to 1 with respect to the case without coding. In particular, Figure 5.18 shows an interesting property: augmenting the redundancy in the pull strategy decreases the overhead to achieve a reliable delivery. This is motivated by the fact that coding determines not only a reduction of the fan-out, as also experienced by the push style, but in this case also a reduction of the number of retransmissions, as depicted in Figure 5.19.

On the contrary, this behavior is not present in the push style (5.17): due to its proactive nature, redundant packets are forwarded as soon as an event arrives to a subscriber. As such, augmenting the redundancy, augments also the overhead.

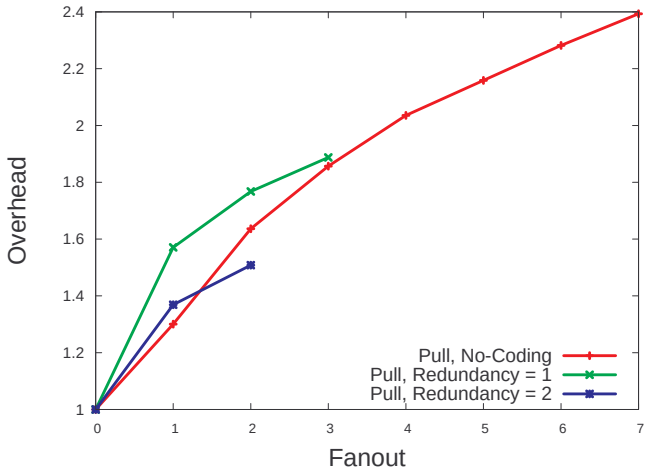


Figure 5.18: Overhead for combined pull gossip and coding when redundancy degree is varied.

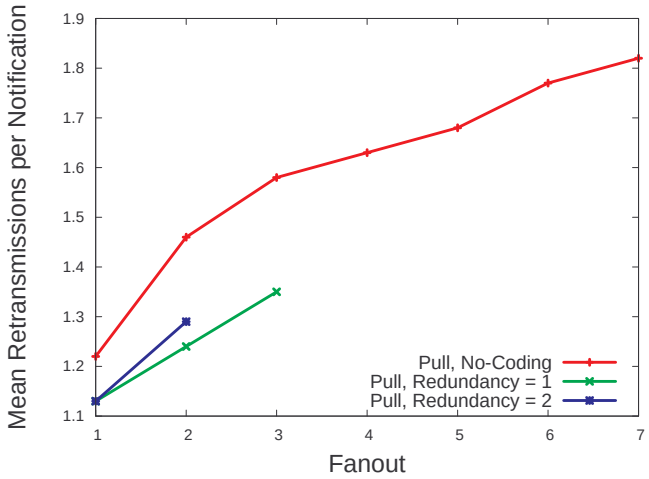


Figure 5.19: Number of needed retransmissions per event in the pull gossip when redundancy degree is varied.

5.5.3 Performance

Figure 5.20 illustrates the mean latency of the gossip approaches to achieve a full success rate by varying the fan-out. As expected, push and push/pull exhibit better performance than pull gossip, due to its periodical dissemination of the last received event identifiers.

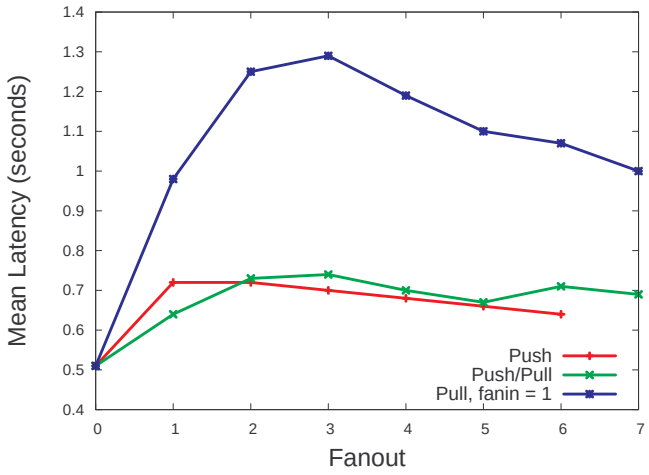


Figure 5.20: Mean latency for the different gossip strategies.

In addition, Figure 5.21 clearly shows that in pull gossip the fan-in further compromises the performance: the higher the fan-in, the greater the delivery latency. This is motivated by the trade-off between fan-in and fan-out: a higher fan-in decreases the fan-out, but the complete reconstruction of an event is spread over more (periodic) gossip executions.

Not surprisingly, applying coding has a good effect on the mean latency, as depicted in Figure 5.22, due to its ability to provide useful packets during an event recovery procedure. This result confirms what we expected from the theoretical analysis: coding is able to reduce the interaction among nodes during the recovery phase of the protocol, so to deliver better performance in terms of latency.

Finally, we analyze the standard deviation of the mean dissemination latency in order to detect possible fluctuations in the event delivery time that can compromise the timeliness requirement. Figure 5.23 illustrates the standard deviation for the push and pull gossip strategies. Due to a reduction of the fan-out and of the number of retransmissions in the pull style, coding is able to decrease the latency standard deviation: with two linear combinations, push and pull experience respectively a reduction of 14% and 19%.

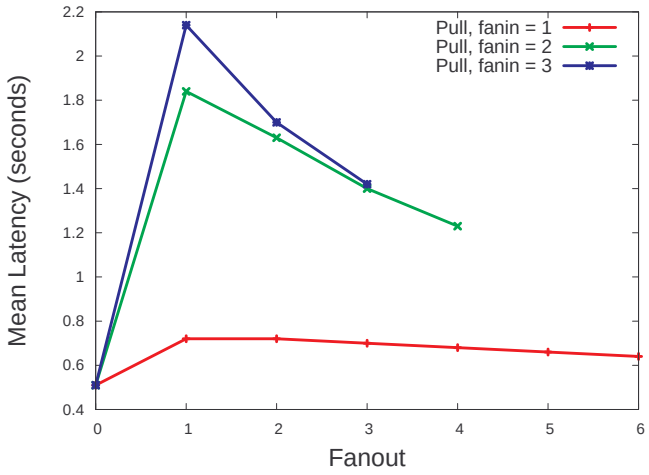


Figure 5.21: Mean latency of pull gossip by varying the fan-out for different fan-in values.

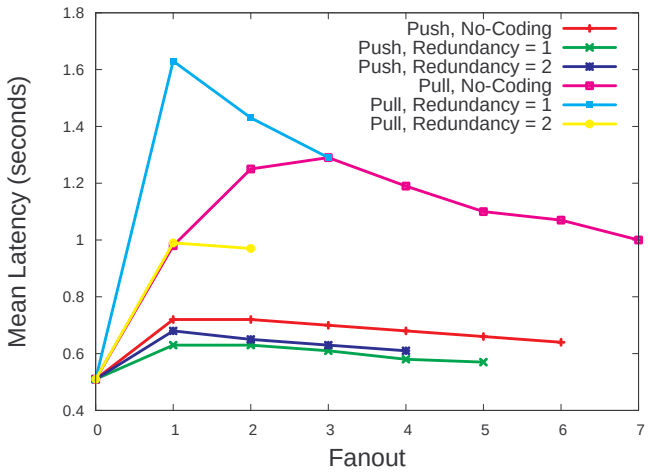


Figure 5.22: Mean latency of the combined approaches by varying the fan-out for a different redundancy degree.

5.5.4 Dependance on network dynamics

To investigate the effect of network dynamics on the obtainable performance, we have compared the mean latency in the push and pull gossip strategies

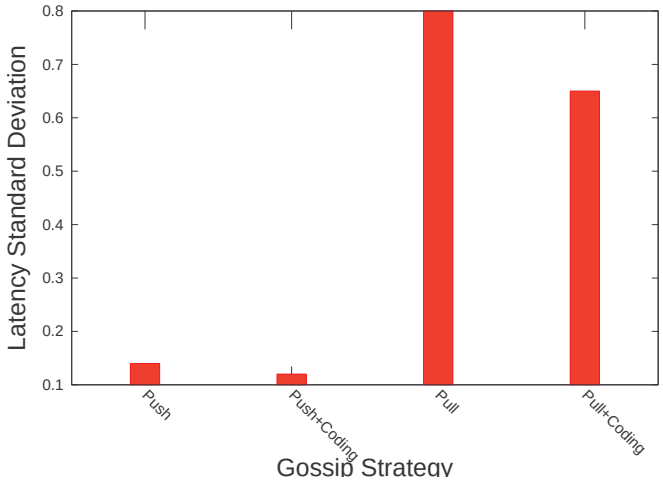


Figure 5.23: Standard deviation of the mean dissemination latency for push and pull gossip, with and without coding.

when PLR is varied. The applied redundancy is composed by two linear combinations. The results of this experiment are reported in Figure 5.24.

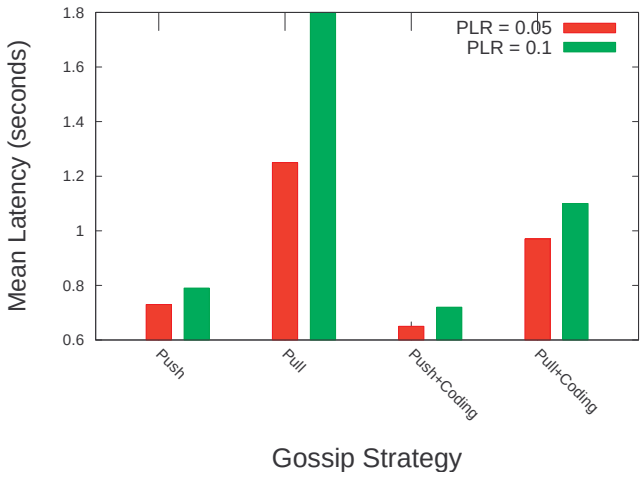


Figure 5.24: Mean dissemination latency for push and pull gossip, with and without coding, by varying the network conditions (i.e., PLR).

Specifically, push and pull strategies show a strong improvement, respectively 8% for push and 31% for pull moving from PLR = 0.05 to PLR = 0.1. In addition, this improvement is also evident for fixed PLR values (i.e., when we separately consider PLR = 0.05 and PLR = 0.1), especially in presence of the pull strategy, even in this case motivated by the reduced number of retransmissions in presence of coding (see Figure 5.19 for reference).

5.5.5 Final considerations

In the experimental analysis we evaluated the impact of coding through a simulation-based study conducted on a real workload taken from the air traffic control scenario. The obtained results state that coding helps to decrease the number of redundant packets (Figure 5.10) and the gossip fan-out (Figures 5.11-5.14) to achieve a reliable delivery of events, due to its ability to provide useful packets during an event recovery procedure. This obviously translates into a reduction of the overhead (Figures 5.15-5.18) and the mean notification latency (Figures 5.20-5.23), even when the network conditions vary (Figure 5.24). In particular, in the pull strategy this result is also obtained by the reduction of the number of retransmissions to fully reconstruct an event (Figure 5.19).

From a practical point of view, we have that an application with strong timeliness requirements should use a push-based gossip recovery strategy with coding: in the considered real workload scenario, we obtain that each event is reliably notified by all subscribers within 500-600 milliseconds. This comes at expenses of a higher overhead than in a pull-based strategy, where its notification latency is however affected by the retransmission period, making this solution suitable just in the presence of less stringent timeliness constraints.

5.6 Related work

Timeliness and reliability in publish/subscribe middleware have been considered just as separate aspects, sometimes resulting in conflict between them, making current solutions not suitable for SoS. As a concrete example, reliability is often ensured by means of retransmissions, i.e., by using TCP links [35, 92] or Automatic Repeat reQuest (ARQ) schemes. In particular, these schemes comprises approaches that detect any possible message omission due to the manifestation of some kind of fault, and ask for a retransmission. Depending on the contacted node for a retransmission, we can have a further classification of the ARQ schemes [68]: *sender-based*, i.e., all the nodes always contact the multicaster; *parent-based*, i.e., a node always contacts its parent

in a multicast tree; *neighbor-based*, i.e., retransmission duties are distributed among any neighboring node. The last one is further divided in the following techniques:

- *Lateral Error Recovery (LER):* all the nodes are randomly grouped in distinct planes. The neighborhood of a node is composed by other nodes in different planes [115].
- *Cooperative Error Recovery (CER):* nodes are clustered in groups whose members are characterized by a negligible loss correlation (i.e., if a node experiences a message loss, it is unlikely that all the nodes loose the same message). Hence, a node selects its neighbors among the members of its group [105].
- *Gossiping algorithms:* [10, 11, 40].

However, retransmission implies that the time needed to deliver an event becomes unpredictable and highly fluctuating, so to violate timeliness, since the number of retransmissions needed to deliver that event depends on the network conditions. In the literature of reliable communications, due to its ability to reduce the delivery time, network coding represents a valid alternative to retransmission. As shown in this Chapter, network coding also provides a low overhead in terms of exchanged messages, other than a potential throughput improvement and a high degree of robustness [56]. To this end, it has been widely applied to data dissemination in large scale systems; several works have shown improvements in the expected file download time in content distribution networks [60], robustness to network changes or link failures [65] and resiliency against random packet loss and delay [37].

In this Chapter, after introducing how network coding works and theoretically showing its benefit on the information delivery performance, we investigated on how network coding and gossip can ensure both timeliness and reliability in publish/subscribe systems. Specifically, we have improved a best-effort topic-based publish/subscribe by (i) introducing the possibility of applying coding at the publisher, (ii) using a gossip strategy to recover any lost data at each destination, and (iii) integrating coding within the retransmissions used by gossip.

The most similar work to ours is [8], where coding is combined with gossip at receiver side: coding is used only when gossipers have to retransmit in push mode their received data. We considerably differ from this work in the following ways: (i) we apply coding also at the publisher site, (ii) we investigate several gossip approaches and the benefit that coding can bring to them, and (iii) we evaluate the effects of coding not only on the delivery performance but also for the imposed overhead and in different network conditions. Another

similar work is presented in [43], which differs from ours due to its theoretical nature (i.e., dissemination approaches are only studied by means of analytical models), and its evaluation metrics (i.e., gossip with and without coding is studied only with respect to latency). Finally, authors in [103] propose a combination of FEC and ARQ schemes to limit, with high probability, the packet loss rate of overlay channels to a target vale q. Specifically, the protocol restricts the number of retransmissions to at most one, in order to minimize the end-to-end latency. Packets are divided into windows, each one with a FEC redundancy factor r_1. In a second round (i.e., retransmission), if a window is non-recoverable, a node retransmits the lost packets with a redundancy factor r_2 (r_1 and r_2 are estimated based on the packet loss rate). The main difference with our solution lies in the fact that the protocol presented in [103] is more suitable for streaming applications, where a minimum packet loss rate can be tolerated, while we concentrate on critical systems in which reliable delivery is of paramount importance.

5.7 Future work

In this Chapter we described an algorithm for timely and reliable event dissemination in publish/subscribe middleware by means of network coding and gossip. Although we showed how the joint use of these two approaches improves the performance with respect to a plain protocol, we planned for the close future to further improve our solution with several optimizations. The first aspect that we are currently considering is how to dynamically adapt the redundancy degree based on the network conditions. In our approach coding is both applied at the publisher's site and at subscribers' site when a retransmission is triggered during a gossip recovery procedure. In both cases, the user is free to choose the redundancy degree to be applied. However, it can be possible to properly tune this degree when applied by a subscriber during the recovery phase of the protocol, so to increase the probability that the retransmission is able to completely recover the lost event. The tuning mechanism depends on the current network conditions; to this end, we are studying a mathematical model that analyzes the mean number of lost packets to infer the proper redundancy degree for a given subscriber.

Another aspect we are considering is how to select gossip partners during the recovery phase. In this Chapter we assumed a random uniform selection mechanism; however, we can improve the efficiency of such a solution by selecting nodes with proper heuristics. To this end, we plan to use a *polarized* gossip, in which a subscriber assignes a weight to a subset of other subscribers interested in the same topic returned by the peer sampling service. The weight assignement follows the probability that the returned subscriber has a missing

packet. Hence, a subscriber contacts f of the highest-weighted subscribers returned by the peer sampling service, i.e., those with the highest probability to have a missed packet. In particular, we plan to devise two different models for the weight assignement: one based on the current network status, which requires to infer the mean number of lost packets for each incoming overlay link, and another one based on the position of nodes in the diffusion overlay, with the idea that nodes closer to the source of information have a higher probability to have a packet.

Chapter 6

Conclusion

In the last few years, the publish/subscribe paradigm gained attention as a middleware platform to connect geographically sparse systems into System of Systems (SoS). Power grids, transport infrastructures (airports and seaports), financial infrastructures, next generation intelligence platforms, are example of SoS, where the information produced by multiple sources has to be delivered to multiple destinations distributed across the world. The publish/subscribe paradigm is appealing for its intrinsic space, time and synchronization decoupling properties, that satisfy the scalability requirement of SoS. Research prototypes as well as commercial systems have been recently proposed to be employed in several contexts. However, although many real world applications require support for a broader set of QoS aspects, other than scalability, the majority of publish/subscribe systems either addresses those aspects in a small scale system, or operates only on a best-effort basis. Middleware solutions such as DDS, TIBCO Rendezvous are used to provide high performance and reliable delivery in local systems, while they degrade performance when applied in large scale environments. In addition, solutions such as SCRIBE, IndiQoS, JEDI fit only partially the QoS requirement of large scale SoS, providing reliability, timeliness and ordering singularly and not at once.

In this thesis, we proposed to fill the gap between application requirements and publish/subscribe QoS properties by devising novel algorithms to address ordering, timeliness and reliability issues. Specifically, in Chapter 1 we defined several QoS policies for delivery (best effort, reliable), ordering (FIFO, causal, total, real time) and timeliness, and, then, we analyzed several case studies, namely collaborative security, stock market, active database in cloud computing and air traffic control to evidence which of those policies had to be considered in these different contexts. From this analysis we highlighted the

current gap between application requirements and middleware guarantees in terms of QoS, by also discussing the challenges in the design of algorithms for large scale systems.

The main contribution of the thesis was to devise a framework to address ordering, timeliness and reliability in SoS that could be posed on top of a generic topic-based publish/subscribe middleware, and it is described in Chapters 3 and 5. Specifically, in Chapter 3 we proposed an algorithm for out-of-order notification detection that guarantees that events published on different topics will be either delivered in the same order to all the subscribers of those topics, or tagged as out-of-order. Our algorithm is completely distributed and relies on a sequencing network of topic managers that collaboratively generate a timestamps for each published event. A deterministic total ordering relation among topics determines a one-way sequence of topic managers that insert a sequence number that represents the number of events previously published on their topics. This timestamp is then used on subscribers' side to infer the correct order of events. In addition, we provided a dynamic algorithm to modify at run-time the topic ordering relation, in order to adapt the current publication popularity. In the experimental analysis we showed how this dynamic adaptation is useful to decrease the notification latency, the bandwidth overhead due to the use of timestamps and the load on topic managers imposed by the timestamp generation. We also developed a prototype of the algorithm that has been evaluated in comparison to a solution based on JGroups, showing that our solution performs better in presence of a high publication rate. In addition, in Chapter 4 we applied the ordering algorithm to the context of database replication in cloud computing, and compared it with a solution that ensures eventual consistency [16]. Our protocol proved to reduce the number of rollbacks to consistently update database replicas even in presence of a high event rate, at the cost of a higher notification latency when the number of topics in the system grows. Finally, the proposed algorithm improves the current state of the art by: (i) scaling over large networks; (ii) avoiding the need of synchronization among interacting parts, as required by the publish/subscribe paradigm; and (iii) dynamically handling subscriptions/unsubscriptions, without requiring to build from skratch the brokering network.

In Chapter 5, instead, we described an algorithm to guarantee both timeliness and reliability in publish/subscribe systems over WAN. Typically, these two properties are either satisfied in local networks ([3, 106]) or considered as separate aspects, sometimes in conflict between them, with reliability obtained by means of retransmissions (so to intoduce delay penalties), or timeliness obtained by softening reliability requirements. We aimed to fill this gap by using two different approaches, namely network coding and gossip, the former known to improve the throughput and to reduce the delivery time in data dissemina-

tion, the latter to improve the reliability in event delivery. We considered the case in which a publisher divides an event in plain packets and sends them through the network. In addition, the publisher sends redundant packets that are linear combinations of the original plain packets. Subscribers can reconstruct the whole information through the reception of plain and coded packets. However, when the received packets are not enough to fully reconstruct an event, a subscriber initiates a gossip procedure by randomly contacting one or more subscribers interested in the same topic. We provided a theoretical analysis that showed the benefit of using coding during the recovery of missed packets in terms of number of gossip interactions and message overhead, with respect to a plain recovery protocol. In addition, we also conducted an experimental analysis on a real workload taken from the air traffic control scenario, that confirmed the theoretical results and evaluated the performance of our algorithm by applying different gossip strategies. Our solution improves the current state of the art by: (i) introducing coding at publishers' site; (ii) investigating several gossiping strategies and the benefit that coding can bring to them; and (iii) evaluating not only notification latency but also the message overhead, by considering different network conditions.

Concluding, let us remark that each proposed solution presents some open problems that we plan to address in the close future (i.e., how to guarantee a causality relation among events published on different topics for the ordering algorithm, or how to improve the timeliness and reliability in event delivery by using a gossiping recovery strategy where partners are selected based on the probability to have a missed packet instead of a random selection). They have been discussed in the future work of the respective Chapters of the thesis.

Bibliography

[1] Corba notification service. http://www.omg.org/corba/. 19

[2] Garr: the italian research and academic network. http://www.garr.it/stampaGARR/materiali/leaflet_RETE_GARR_ENG.pdf. 76

[3] The omg data distribution service. http://portals.omg.org/dds/. 17, 19, 30, 116

[4] Credit card firm hit by DDoS attack. http://www.computerworld.com/securitytopics/security/story/0,10801,96099,00.html, 2010. 5

[5] FBI investigates 9 Million ATM scam. http://www.myfoxny.com/dpp/news/090202_FBI_Investigates_9_Million_ATM_Scam, 2010. 5

[6] L. Aniello, G. Lodi, and R. Baldoni. Inter-domain stealthy port scan detection through complex event processing. In *Proceedings of the 13th European Workshop on Dependable Computing*, pages 67–72. ACM, 2011. 6

[7] L. Aniello, G. A. Di Luna, G. Lodi, and R. Baldoni. A collaborative event processing system for protection of critical infrastructures from cyber attacks. In *to appear in 30th Conference on System Safety, Reliability and Security*, Napoli, September 2011. 6

[8] M. Balakrishnan, K.P. Birman, A. Phanishayee, and S. Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast. *Proceedings of the 4th USENIX Symposium on Networked System Design & Implementation*, pages 73–86, April 2007. 112

[9] R. Baldoni, R. Beraldi, G. Lodi, M. Platania, and L. Querzoni. Moving core services to the edge in ngns for reducing managed infrastructure size. In *CNSM*, pages 410–413, 2010. 24

[10] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni. Tera: topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 2–13. ACM, 2007. 112

[11] R. Baldoni, R. Beraldi, L. Querzoni, G. Cugola, and M. Migliavacca. Content-based routing in highly dynamic mobile ad hoc networks. *International Journal of Pervasive Computing and Communications*, 1(4):277–288, 2005. 112

[12] R. Baldoni, S. Bonomi, G. Lodi, M. Platania, and L. Querzoni. Data dissemination supporting complex event pattern detection. *International Journal of Next Generation Computing*, 2011. 24

[13] R. Baldoni, S. Bonomi, M. Platania, and L. Querzoni. Dynamic message ordering for topic-based publish/subscribe systems. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2012. 24

[14] R. Baldoni, S. Cimmino, and C. Marchetti. A classification of total order specifications and its application to fixed sequencer-based implementations. *J. Parallel Distrib. Comput.*, 66(1):108–127, 2006. 31

[15] R. Baldoni, M. Contenti, S. Tucci-Piergiovanni, and A. Virgillito. Modeling publish/subscribe communication systems: towards a formal approach. In *Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on*, pages 304–311. IEEE, 2003. 4

[16] R. Baldoni, R. Guerraoui, R. Levy, V. Quema, and S. Tucci-Piergiovanni. Unconscious eventual consistency with gossips. *Stabilization, Safety, and Security of Distributed Systems*, pages 65–81, 2006. 23, 72, 75, 76, 78, 116

[17] R. Baldoni, C. Marchetti, and A. Virgillito. Impact of WAN Channel Behavior on End-to-end Latency of Replication Protocols. In *Proceedings of the Sixth European Dependable Computing Conference*, page 118. IEEE Computer Society, 2006. 52

[18] R. Baldoni, M. Platania, L. Querzoni, and S. Scipioni. A peer-to-peer filter-based algorithm for internal clock synchronization in presence of corrupted processes. In *PRDC*, pages 64–72, 2008. 24

[19] R. Baldoni, M. Platania, L. Querzoni, and S. Scipioni. Practical uniform peer sampling under churn. In *ISPDC*, 2010. 24

[20] R. Baldoni and M. Raynal. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*, 3(2), 12 2002. 67

[21] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *DIS, Universita di Romaâ La Sapienzaâ, Tech. Rep*, 2005. 16

[22] D. Basin, K. Birman, I. Keidar, and Y. Vigfusson. Sources of instability in data center multicast. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, pages 32–37. ACM, 2010. 18

[23] I. Baumgart, B. Heep, and S. Krause. OverSim: A Scalable and Flexible Overlay Framework for Simulation and Real Network Applications. *Proceedings of the 9th International Conference on Peer-to-Peer Computing (IEEE P2P 09)*, pages 87–88, September 2009. 100

[24] S. Behnel, L. Fiege, and G. Muehl. On quality-of-service and publish-subscribe. In *Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on*, pages 20–20. IEEE, 2006. 19

[25] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 5. Addison-wesley New York, 1987. 73, 76

[26] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 3–9, 2002. 29

[27] S. Bhola, R. Strom, S. Bagchi, Z. Yuanyuan, and J. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 02)*, pages 7–16, June 2002. 81

[28] K. Birman. Rethinking multicast for massive-scale platforms. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 1–1. IEEE, 2009. 22

[29] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009. 11, 12, 16, 18, 71, 74

[30] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987. 46

[31] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993. 49

[32] F. Cao and J. Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. *Middleware 2005*, pages 292–313, 2005. 17

[33] N. Carvalho, F. Araujo, and L. Rodrigues. Scalable qos-based event routing in publish-subscribe systems. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 101–108. IEEE, 2005. 17, 18, 19

[34] 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. 17

[35] M. Castro, P. Druschel, A.M. Kermarrec, and A.I.T. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002. 17, 19, 32, 111

[36] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996. 66

[37] P.A. Chou, Y. Wu, and K. Jain. Practical network coding. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, volume 41, pages 40–49. The University; 1998, 2003. 112

[38] Comifin. CoMiFin - Communication Middleware for Monitoring Financial Critical Infrastrucures. http://www.comifin.eu/, 2008. 5, 20

[39] A. Corsaro, L. Querzoni, S. Scipioni, S. Tucci-Piergiovanni, and A. Virgillito. Quality of service in publish/subscribe middleware. *Global Data Management*, 2006. 19, 20

[40] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. *Proceeding of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 04)*, pages 552–561, March 2000. 112

[41] Paolo Costa and Gian Pietro Picco. Semi-probabilistic content-based publish-subscribe. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 575–585, Washington, DC, USA, 2005. IEEE Computer Society. 85

[42] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, pages 827–850, 2001. 17, 18

[43] S. Deb, M. Medard, and C. Choute. Algebraic gossip: a network coding approach to optimal multiple rumor mongering. *IEEE Transactions on Information Theory*, 52(6):2486–2507, June 2006. 113

[44] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007. 22

[45] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004. 12, 31, 66

[46] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987. 85

[47] E.O. Elliott. Estimates of Error Rates for Codes on Burst-Noise Channels. *Bell System Technical Journal*, 42:1977–1997, 1963. 82

[48] E.N. Elnozahy. On the relevance of communication costs of rollback-recovery protocols. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 74–79. ACM, 1995. 72

[49] C. Esposito. Data Distribution Service (DDS) Limitations for Data Dissemination w.r.t. Large-scale Complex Critical Infrastructures (LCCI). *Mobilab Technical Report (www.mobilab.unina.it)*, March 2011. 100

[50] C. Esposito, S. Russo, R. Beraldi, and M. Platania. On the benefit of network coding for timely and reliable event dissemination in wan. In *1st International Workshop on Network Resilience: From Research to Practice*, 2011. 24

[51] C. Esposito, S. Russo, R. Beraldi, M. Platania, and R. Baldoni. Achieving reliable and timely event dissemination over wan. In *13th International Conference on Distributed Computing and Networking*, 2012. 24

[52] 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. 2, 16

[53] E. Fidler, H.A. Jacobsen, G. Li, and S. Mankovski. The padres distributed publish/subscribe system. *Feature Interactions in Telecommunications and Software Systems, VIII*, 2005. 17

[54] L. Fiege and G. Muehl. Rebeca event-based electronic commerce architecture, 2000. 17, 19

[55] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Journal Discrete Applied Mathematics, Volume 39 Issue 3, Nov. 11, 1992*, November 1992. 93

[56] C. Fragouli, J. Le Boudec, and J. Widmer. Network coding: an instant primer. *Computer Communication Review*, 36(1):63, 2006. 84, 86, 87, 104, 112

[57] FreePastry. http://www.freepastry.org/. 52

[58] H. Garcia-Molina and A.M. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems (TOCS)*, 9(3):242–271, 1991. 29, 30, 66

[59] E.N. Gilbert. Capacity of a Burst-Noise Channel. *Bell System Technical Journal*, 39:1253–1265, 1960. 82

[60] C. Gkantsidis and P.R. Rodriguez. Network coding for large scale content distribution. In *Proceedings of the IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2235–2245, 2005. 112

[61] A.S. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 110–123. Springer-Verlag, 1989. 66

[62] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997. 73

[63] G. Hasslinger and O. Hohlfeld. The Gilbert-Elliott Model for Packet Loss in Real Time Services in the Internet. *Proceedings of the 14th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, pages 1–15, March-April 2008. 83

[64] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. 73

[65] T. Ho, M. Médard, R. Koetter, D.R. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006. 112

[66] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, 2007. 86

[67] JGroups. http://www.jgroups.org/. 23, 30

[68] X. Jin, W. P. Ken Yiu, and S. H. Gary Chan. Loss Recovery in Application-Layer Multicast. *IEEE MultiMedia*, 15(1):18–27, January 2008. 111

[69] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of algorithms*, 11(3):462–491, 1990. 72

[70] T.T.Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 454–461. IEEE, 1991. 72

[71] R.S. Kazemzadeh and H.-A. Jacobsen. Reliable and Highly Available Distributed Publish/Subscribe Service. *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems (SRDS 09)*, pages 41–50, September 2009. 81

[72] R.S. Kazemzadeh and H.A. Jacobsen. Partition-tolerant distributed publish/subscribe systems. In *The 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 101–110. IEEE, October 2011. 66, 67

[73] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 424–431. IEEE, 1999. 72

[74] A.-M. Kermarrec, L-Massoulié, and A. J. Ganesh. Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 14(2):1–11, February 2003. 85

[75] A. Konrad, B.Y. Zhao, and A.D. Joseph. Determining Model Accuracy of Network Traces. *Journal of Computer and System Sciences*, 72(7):1156–1171, 2006. 83

[76] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 47, 66, 67

[77] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001. 71

[78] S.Y.R. Li, R.W. Yeung, and N. Cai. Linear network coding. *Information Theory, IEEE Transactions on*, 49(2):371–381, 2003. 82, 84

[79] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *Cooperative Information Systems, 1999. CoopIS'99. Proceedings. 1999 IFCIS International Conference on*, pages 70–78. IEEE, 2002. 22, 25, 29, 30, 68

[80] Y. Liu and B. Plale. Survey of publish subscribe event systems. *Computer Science Dept, Indian University*. 16

[81] G. Lodi, L. Querzoni, R. Baldoni, M. Marchetti, M. Colajanni, V. Bortnikov, G. Chockler, E. Dekel, G. Laventman, and A. Roytman. Defending financial infrastructures through early warning systems: the intelligence cloud approach. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, page 18. ACM, 2009. 5

[82] C. Lumezanu, N. Spring, and B. Bhattacharjee. Decentralized message ordering for publish/subscribe systems. *Middleware 2006*, pages 162–179, 2006. 29, 30, 66

[83] S.P. Mahambre, M. Kumar, and U. Bellur. A taxonomy of qos-aware, adaptive event-dissemination middleware. *Internet Computing, IEEE*, 11(4):35–44, 2007. 17, 19

[84] A. Malekpour, A. Carzaniga, G. Toffetti Carughi, and F. Pedone. Probabilistic fifo ordering in publish/subscribe networks. Technical report, Faculty of Informatics, University of Lugano, Technical Report USI-INF-TR-2011-2, 2011. 29

[85] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. Characterization of Failures in an Operational IP Backbone Network. *IEEE/ACM Transactions on Networking (TON)*, 16(4):749–762, August 2008. 81

[86] L. Massoulié, E. Le Merrer, A.M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 123–132. ACM, 2006. 86

[87] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989. 67

[88] G. Muehl, L. Fiege, and P. R. Pietzuch. *Distributed event-based systems*. Springer-Verlag, 2006. 2, 17

[89] Oracle. Oracle streams advanced queuing. http://download.oracle.com/docs/cd/B19306_01/server.102/b14257/aq_intro.htm. 17, 19

[90] M. Palmer and M. Dzmuran. An Introduction to Event Processing, white paper. http://www.cnetdirectintl.com/direct/fr/2009/progress/0907_centre_ressources/ressources/1-1_LB_UK_introduction_CEP.pdf, 2009. 8

[91] P. R. Pietzuch, D. Eyers, S. Kounev, and B. Shand. Towards a common api for publish/subscribe. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 152–157. ACM, 2007. 28

[92] P.R. Pietzuch and J.M. Bacon. Hermes: A distributed event-based middleware architecture. 2002. 17, 19, 81, 111

[93] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, 2004. 29

[94] M. Platania, R. Beraldi, G. Lodi, L. Querzoni, and R. Baldoni. Supporting ngns core software services: A hybrid architecture and its performance analysis. *Journal of Network and Systems Management*, pages 1–19, 2011. 24

[95] S. K. Rao. Algorithmic trading: Pros and cons. http://www.tcs.com/SiteCollectionDocuments/WhitePapers/TCS_FS_algorithmictrading.pdf, 2010. 8

[96] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001. 51

[97] F.B. Schneider. Replication management using the state-machine approach. *Distributed systems*, 2:169–198, 1993. 49

[98] Single European Sky ATM Research (SESAR). http://www.sesarju.eu/about. 2

[99] R. Shoup. Architectural principles. http://www.infoq.com/presentations/shoup-ebay-architectural-principles. 11

[100] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 223–238. ACM, 1989. 72

[101] 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. 17

[102] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985. 72

[103] L. Subramanian, I. Stoica, H. Balakrishnan, and R.H. Katz. Overqos: An overlay based architecture for enhancing internet qos. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation-Volume 1*, pages 6–21. USENIX Association, 2004. 113

[104] Inc. Sun Microsystems. Java message service (jms) specification 1.3.1. http://download.oracle.com/javaee/1.3/jms/tutorial/. 17, 19

[105] G. Tan, S. A. Jarvis, and D. P. Spooner. Improving the Fault Resilience of Overlay Multicast for Media Streaming. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(6):721–734, June 2007. 112

[106] Inc. TIBCO. Tibco rendezvous. http://www.tibco.com/products/soa/messaging/default.jsp. 17, 19, 116

[107] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003. 85

[108] A. Varga. The OMNET++ Discrete Event Simulation System. *Proceedings of the European Simulation Multiconference (ESM 01)*, pages 319–324, June 2001. 100

[109] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009. 12, 22, 71, 74

[110] WANem. http://wanem.sourceforge.net/. 52

[111] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann Pub, 1996. 11

[112] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *srds*, page 206. Published by the IEEE Computer Society, 2000. 74

[113] L. Wilhelmsson and L.B. Milstein. On the effect of imperfect interleaving for the Gilbert-Elliott channel. *IEEE Transactions on Communications, Volume - 47 Issue:5, May 1999*, May 1999. 96

[114] G.A. Wilkin and P. Eugster. Multicast with aggregated deliveries. In *Proceedings of the 1st International Workshop on Algorithms and Models for Distributed Event Processing (AlMoDEP)*. ACM, 2011. 67

[115] W.-P. K. Yiu, K.-F. S. Wong, S.-H. G. Chan, W.-C. Wong, Q. Zhang, W.-W. Zhu, and Y.-Q. Zhang. Lateral Error Correction for Media Streaming in Application-Level Multicast. *IEEE/ACM Transactions on Multimedia (T-MM)*, 8(2):219–232, April 2006. 112

[116] X. Yu, J. W. Modestino, and X. Tian. The Accuracy of Gilbert Models in Predicting Packet-Loss Statistics for a Single-Multiplexer Network Model. *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 4:2602–2612, March 2005. 83

[117] G. Zhang and M. Parashar. Cooperative detection and protection against network attacks using decentralized information sharing . *Cluster Computing*, 13(1):67–86, 2010. 85

[118] K. Zhang, V. Muthusamy, and H.A. Jacobsen. Total order in content-based publish/subscribe systems. Technical report, University of Toronto, 2010. 66, 67