A Horizontally Scalable and Reliable Architecture for Location-based Publish-Subscribe

Bertil Chapuis
Université de Lausanne
bertil.chapuis@unil.ch

Benoît Garbinato
Université de Lausanne
benoît.garbinato@unil.ch

Lucas Mourot
EPFL
lucas.mourot@epfl.ch

***Abstract*—With billions of connected users and objects, location-based services face a massive scalability challenge. We propose a horizontally-scalable and reliable location-based publish/subscribe architecture that can be deployed on a cluster made of commodity hardware. As many modern location-based publish/subscribe systems, our architecture supports moving publishers, as well as moving subscribers. When a publication moves in the range of a subscription, the owner of this subscription is instantly notified via a server-initiated event, usually in the form of a push notification. To achieve this, most existing solutions rely on classic indexing data structures, such as R-trees, and they struggle at scaling beyond the scope of a single computing unit. Our architecture introduces a multi-step routing mechanism that, to achieve horizontal scalability, efficiently combines range partitioning, consistent hashing and a min-wise hashing agreement. In case of node failure, an active replication strategy ensures a reliable delivery of publication throughout the multistep routing mechanism. From an algorithmic perspective, we show that the number of messages required to compute a match is optimal in the execution model we consider and that the number of routing steps is constant. Using experimental results, we show that our method achieves high throughput, low latency and scales horizontally. For example, with a cluster made of 200 nodes, our architecture can process up to 190'000 location updates per second for a fleet of nearly 1'900'000 moving entities, producing more than 130'000 matches per second.**

***Index Terms*—Internet of Things; Publish subscribe systems; Distributed databases; Spatial databases; Scalability; Fault tolerance**

I. Introduction

Today, moving objects and moving users produce massive amounts of geo-located data. For example, it is now common for cellular network operators and data analytics companies to collect up to several millions of geographical data points per seconds. Furthermore, this trend is not likely to stop: according to Gartner and IDC,[1] there will be between 25 to 30 billion connected objects by 2020. The emergence of this ecosystem of connected objects, known as the *Internet of Things* (IoT), opens new opportunities, but also comes with new challenges in terms of development and deployment.

A. Location-Based Publish-Subscribe

Here the location-based publish-subscribe paradigm is of particular interest for developing mobile applications that want to take advantage of the IoT ecosystem. With this communication paradigm, connected objects are able to issue publications and subscriptions that are geographically scoped and that move with them. The scope of a publication or a subscription is known as its *space*. A match occurs between a given publication and a given subscription if both a *content* criterion and a *context* criterion are met simultaneously. The content criterion expresses a semantic relationship between the publication and the subscription, as captured by traditional publish-subscribe systems such as the Java Messaging Service API.[2] The context criterion then expresses some proximity condition between the publication space and subscription space, hence the term *location-based* or sometimes *location-aware publish-subscribe* [6], [16].

The location-based publish-subscribe paradigm is of great interest for any mobile application that requires a precise and up-to-date knowledge of the context of its users. Therefore, it could be used to improve user experience in various domains including social networking, transportation, video game, and augmented reality. Although this communication paradigm offers great expressiveness and flexibility, scaling its implementation to billions of objects is far from trivial. As of today, location-based publish/subscribe solutions described in the literature have addressed the scalability problem *vertically*, i.e., with a centralized computing unit responsible for the full workload. Obviously, the vertical scalability approach can only work up to a certain load, that the exponential growth of the IoT ecosystem can easily exceed.

B. Achieving Horizontal Scalability

The traditional centralized spatial indexing approaches mentioned above aim at taking advantage of high stability in geographical locality (because most objects, such as buildings, shops, landmarks, etc., are not moving), via tree-like data structures. Indeed, such approaches are known to be efficient in contexts where the majority of indexed objects fit on a single machine and are static, i.e., when reads on the index greatly outnumber writes. Keeping the underlying tree-based data structures balanced can be very costly in the presence of numerous writes. Therefore, in order to support a large number of moving objects we need to scale out.

Yet to our knowledge, while many efficient spatial data-structures have been proposed to accelerate the computation of

[1] http://www.gartner.com/newsroom/id/3165317
https://www.idc.com/getdoc.jsp?containerId=US40755816

[2] https://jcp.org/en/jsr/detail?id=368

978-1-5386-1679-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SRDS.2017.16

matches between moving publishers and moving subscribers, such as variants of R-trees for instance, the problem has not yet been addressed in terms of *horizontal scalability*, i.e, with many distributed computing units, each one being responsible for only a part of the workload. Here, we consider the problem of horizontally scaling the location-based publish/subscribe communication paradigm. Our starting point consists in fragmenting locality using the notions of range partitioning, in conjunction with consistent-hashing, in order to dynamically distribute the computation of the matches.

C. Contributions & Roadmap

Our key contributions are organized as follows.

1) In Section II, we describe a model for reasoning about the location-based publish/subscribe paradigm in a distributed context and we introduce the problem of scaling out systems supporting this paradigm.
2) In Section III, we introduce a scalable and reliable location-based publish/subscribe architecture. This architecture scales horizontally due to the counter-intuitive idea that selectively fragmenting locality with a combination of range partitioning, consistent hashing and an a priori min-wise hashing agreement can help us compute matches efficiently. In addition, an active replication strategy makes this distributed architecture tolerant to node failures.
3) In Sections IV and V, we evaluate our solution both theoretically and experimentally. For the latter, we implement and evaluate our protocol on a real cluster setup and highlight its performances in terms of horizontal scalability, throughput and latency.

We then conclude this paper by discussing related work in Section VI and future research opportunities in Section VII.

II. Scaling location-based publish-subscribe

We consider a distributed system composed of mobile client nodes, that represent computing devices moving in the field, and of fixed server nodes that represent computing resources in some data center.

A. Client-Side Model

On the client side, mobile nodes can issue long-lived geographically-scoped publications and subscriptions that move with their issuers. Formally, a publication *pub* is defined as tuple $pub = (id, Z, A)$, where $id \in \mathbb{N}$ uniquely identifies *pub*, $Z \in \mathscr{Z}$ denotes the geographical zone[3] where *pub* is active and set $A = \{a_1, a_2, \ldots, a_{|A|}\}$ denotes a collection of attributes of the form $a = (name, value)$. In other words, A defines the *content* of *pub*, whereas Z defines its *context*. Similarly, a subscription *sub* is defined as tuple $sub = (id, Z, A, issuer)$, where *issuer* uniquely identifies the mobile client node that issued *sub*. As publications and subscriptions move with their issuers, the geographical boundaries where they are active are also moving. Therefore, given a moving subscription *sub*

[3]Here $\mathscr{Z}$ denotes the set of all zones definable on the earth surface.

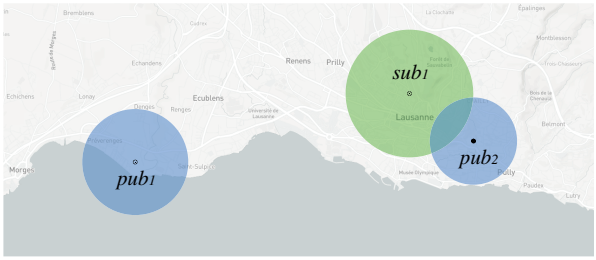


Fig. 1: Example of location match

and a moving publication *pub*, we say that a match occurs when the following two conditions are met: $sub.Z \cap pub.Z \neq \varnothing$ and $sub.A \subseteq pub.A$. When such a match occurs, the mobile node that issued *sub* is notified by asynchronously receiving a tuple (pub, sub). It is worth noting that every time a publication moves within the range of a matching subscription, the subscription's issuer will receive an update.

Intuitively, the first condition captures the fact that the geographical zones of *pub* and *sub* overlap; this is known as a *context match* or a *location match*. Figure 1 depicts two publications pub_1 and pub_2 and a subscription sub_1 that only matches with pub_2 in terms of location. The second condition captures the fact that publication *pub* contains at least all the attributes of subscription *sub*; this is known as a *content match*.

The above client-side model is similar to the one used in [16]. There exists of course many alternative ways to define the notion of content match, but this is out of scope here: in this paper, we focus exclusively on location matches. Similarly, we could define the notion of location match by using an alternative proximity criterion but, as we will see, this would not affect the generality of our approach.

B. Server-Side Model

On the server side, fixed server nodes consist in interconnected virtual or physical nodes running on commodity hardware and organised as a cluster in some data centers. In this paper, we consider a set of distributed processes $\Pi = \{p_1, p_2, \cdots, p_{|\Pi|}\}$ running on server nodes. Furthermore, we assume that processes in Π know each other and communicate by reliably exchanging uniquely identified messages.

C. Scaling horizontally

Using the above client-side and server-side models, we can now specify the horizontal scalability problem. We begin by defining how the two models work together, i.e., how nodes on one side communicate with nodes on the other side.

When a client-side mobile node wants to issue a new publication or a new subscription, or when it moves and hence needs to update the geographical zones associated with its existing publications and subscriptions, it sends messages to the cluster. The latter is responsible for computing the matches that are then sent back to the mobiles nodes that issued the subscriptions concerned by those matches. It is worth noting

that mobile nodes never communicate directly with each others but always do so via cluster nodes.

Here, we address the problem of distributing the computation of matches among cluster nodes in a way that enables the overall system to *scale horizontally*. That is, we want to answer the following question: How can we organize the work of server-side nodes so that by simply adding new nodes to the cluster, we can manage a growing number of moving publications and subscriptions, while maintaining low latency in term of match computation and delivery?

III. A Horizontally Scalable and Reliable Architecture

In this section, we describe a horizontally scalable architecture that supports the location-based publish and subscribe communication paradigm. In contrast to traditional centralized spatial data structures that aim at taking advantage of geographical locality, our solution fragments locality by using the notions of *range partitioning*, in conjunction with *consistent-hashing*, in order to dynamically distribute the computation of matches across sets of processes.

A. Range Partitioning

Map services such as Google Maps[4] or Mapbox[5] typically rely on a grid layout that divides the world into a set of tiles. In this paper, we use tiles to create range partitions along two dimensions. We then use these partitions to distribute the computation of context matches between publications and subscriptions among processes running in our cluster. For this, we introduce set $G = \{t_1, t_2, \ldots, t_{|G|}\}$: it denotes a grid layout on the earth surface consisting in a set of uniquely identified tiles. In addition, we introduce function $tiles: \mathscr{Z} \rightarrow 2^G$ that maps a geographical zone $Z \in \mathscr{Z}$ to its overlapping set of tiles $T_Z \subseteq G$, i.e., we have $T_Z = \{t \in G \mid t \cap Z \neq \varnothing\}$.

Figure 2 illustrates how the *tiles* function is used to determine the sets of tiles overlapping with the publications and subscription depicted in Figure 1. Obviously, rectangle-based grids rely on a map projection, which is necessary for mapping coordinates on a sphere to coordinates on a plane. Such map projections are known to introduce spatial distortions that result in the tiles being not uniform in terms of shape and size. However, as in our context tiles are used as units of distribution and parallelism, these distortions have virtually no effect on performance.

B. Consistent Hashing

Distributed hash tables typically rely on a family of functions that offers *consistent hashing* in order to partition data items across a cluster of processes and to scale horizontally. In other words, the unique identifier of each data item is passed to a *consistent hashing function* and the resulting hash value is then used to find the process responsible for handling that particular data item.

[4]https://maps.google.com
[5]https://www.mapbox.com

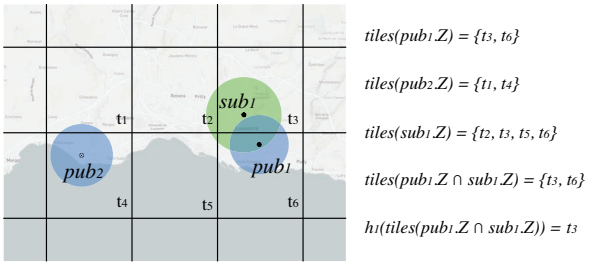


Fig. 2: Example of using the *tiles* function

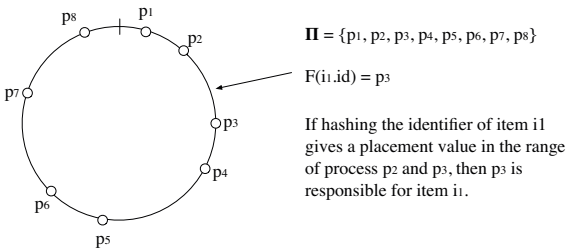


Fig. 3: Distributing data items across processes in a ring

Formally, consistent hashing can be expressed as function $F : I \rightarrow \Pi$ that distributes a set of data items $I = \{i_1, i_2, \ldots, i_{|I|}\}$ across a set of processes Π. As illustrated in Figure 3, it is common to think about consistent hashing as a ring in which the largest possible hash value convolutes to the smallest possible hash value [5]. Each process is assigned a fixed position on the ring, for example by hashing the unique identifier of that process. In order to find the process responsible for handing a given item, the identifier of that item is hashed, then the first process on the ring with a placement value greater than the resulting hash value is selected. Here the monotonicity of F is particularly interesting as it ensures that when processes are added or deleted, the distribution of items across existing processes does not change [14]. However, as illustrated in Figure 3, positioning processes by hash values can lead to a non-uniform distribution of the load on the ring. As a consequence, we rely on a notion similar to the "virtual nodes" used by Dynamo [5]. When processes are added to the system, each of them receives multiple positions in the ring, improving the uniformity of the load distribution.

C. Min-wise Hashing Agreement

Our architecture relies on the tiling of the earth's surface and on a set of consistent hashing functions to partition and distribute the load of computing matches between publications and subscriptions. That is, the subdivision of the earth's surface into tiles is used as a range partitioning criteria, whereas consistent hashing functions are responsible for distributing the load by routing messages to processes. As depicted in Figure 2, a problem occurs when the boundaries of a publication or a subscription overlap with several tiles. In such a case, the same match will be computed on several tiles, here t_3 and t_6, resulting in duplicated messages. A straightforward solution would

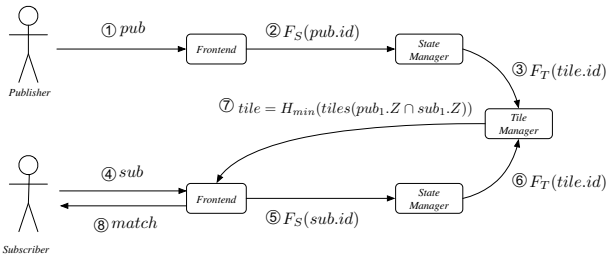


Fig. 4: Overview of a Match Triggering Graph

be to address this issue with an a posteriori agreement, i.e, an centralised process for identifying and eliminating duplicates. However, such a solution would have several disadvantages: First, the detection of duplicates by using a list or a set in an unbounded message stream is not practical due to memory constraints; second, when publications and subscriptions overlap with many tiles, the amount of duplicated messages transmitted in the cluster might result in network congestion.

To avoid these problems completely, we present an efficient a priori min-wise hashing agreement that does not require a centralized coordination. As illustrated in Figure 2, both tiles t_3 and t_6 identify the intersection between pub_1 and sub_1 and trigger a match. Hence, by computing the set $tiles(pub_1.Z \cap sub_1.Z)$, each tile is able to infer which other tile will compute the exact same match. Given this fact, a convention can be used to determine which tile is responsible for sending the match to the end user. Let H be a hash function that maps tiles to distinct integers. Given any set of tiles T, we define $H_{min}(T)$ to be the tile $t \in T$ with the minimum hash value. On this basis, the a priori min-wise hashing agreement can be expressed with one condition: given any $tile$ overlapped by both pub_1 and sub_1, a $tile$ transmit the match to the end user only if the condition $tile = H_{min}(tiles(pub_1.Z \cap sub_1.Z))$ is satisfied.

D. Detailed Architecture and Algorithms

In this section, we provide a more detailed description of the internals of our architecture.

Process roles. When participating in match computation, processes can have one of the three roles described hereafter. When new nodes are added to the cluster to scale out, one or more processes of each role can be started.

1) The *Frontend* role characterizes the set of processes $\Pi_F \subseteq \Pi$ responsible for handling and routing publication and subscription requests in the cluster.
2) The *State Manager* role characterizes the set of processes $\Pi_S \subseteq \Pi$ responsible for tracking and managing the state of publications and subscription in the cluster,
3) The *Tile Manager* role characterizes the set of processes $\Pi_T \subseteq \Pi$ responsible for computing matches between publications and subscriptions that overlap with a specific set of tiles.

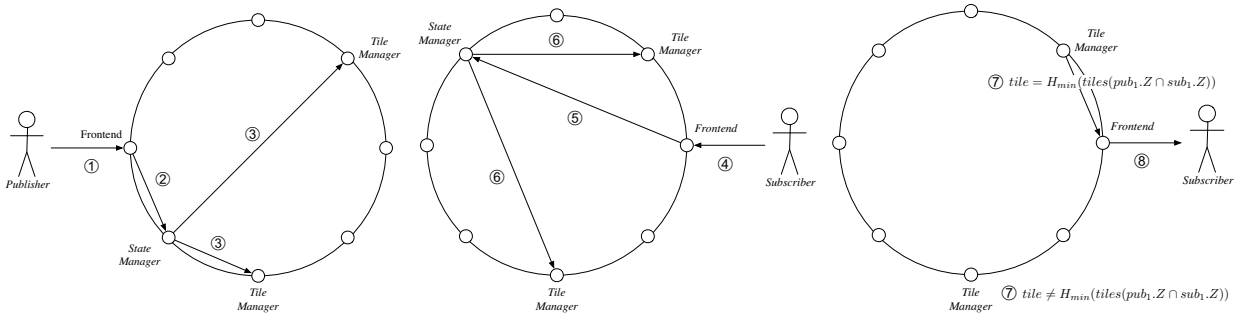
Match Triggering Graph. Figure 4 gives an overview of how the aforementioned concepts all play together. We use the term *Match Triggering Graph* when referring to the graph that contains all the paths of the messages that lead to a match. In other words, a *Match Triggering Graph* corresponds to the paths that link a *State Manager* process responsible for a particular publication, a *State Manager* process responsible for an overlapping subscription and the *Tile Manager* process that computes a match for this publication/subscription pair.

Message routing. Messages are routed across processes by using the two distinct consistent hashing functions:

1) Function $F_S : \mathbb{N} \to \Pi_S$ routes messages to *State Manager* processes by hashing publications and subscription identifiers $id \in \mathbb{N}$.
2) Function $F_T : \mathbb{N} \to \Pi_T$ routes messages to *Tile Manager* processes by hashing tile identifiers $tile.id \in \mathbb{N}$.

Figure 5 illustrates in more details how publications, subscriptions and matches are routed in the cluster. In Figure 5a, the publication is first routed from a *Frontend* process to a *State Manager* process. The *State Manager* is responsible for identifying and notifying the tiles that overlap the geographical zone of a publication. Why is this intermediary step between *Frontend* processes and *Tile Manager* processes necessary? When the location of a publication is updated, it might enter some tiles and leave some others. As a consequence, it is necessary to have a process responsible for tracking the state of a publication in the cluster and notifying *Tile Manager* processes in a consistent manner. In Figure 5b, a similar routing scenario is depicted for subscriptions. As depicted in Figure 5c, the same match can be computed by several *Tile Manager* processes. However, only one of these process satisfies the min-wise hashing condition $tile = H_{min}(tiles(pub_1.Z \cap sub_1.Z))$ and transmits the match to the subscriber. Consequently, the architecture requires no intermediary step for eliminating duplicated matches.

Frontend algorithm. A client is not assumed to know which process of the cluster is responsible for managing the state of a particular publication or subscription. Furthermore, in practice, an implementation of the middleware would typically communicate with subscribers through TCP keep-alive connections. As a consequence, the *Frontend* process is typically used behind a load balancer to parse the protocol-specific requests emitted by clients, to route them to the correct server side *State Manager* processes using consistent hashing, and to forward the computed matches back to the subscribers in the protocol specific format. An hypothetic *Frontend* process can receive four kinds of protocol-specific events from the client: (1) $\langle addPub|pub \rangle$ messages are used to add or update the state of a publication *pub* in the cluster; (2) $\langle deletePub|pub \rangle$ messages are used to delete the state of a publication *pub* from the cluster; (3) $\langle addSub|sub \rangle$ messages are used to add or update the state of a subscription *sub* in the cluster; and (4) $\langle deleteSub|sub \rangle$ messages are used to delete the state of a subscription *sub* from the cluster. In Algorithm 1, the $self$ variable corresponds to the $frontend$ process itself and the $connection$ variable corresponds to a client connection, typically a TCP connection.



(a) Publication routing (b) Subscription routing (c) Match routing

Fig. 5: Distributed routing based on consistent hashing

Algorithm 1 Frontend

upon event $\langle init \rangle$
  $connections \leftarrow \varnothing$

upon event $\langle addPub|pub \rangle$
  send $\langle addPub|pub \rangle$ **to** $F_S(pub.id)$

upon event $\langle deletePub|pub \rangle$
  send $\langle deletePub|pub \rangle$ **to** $F_S(pub.id)$

upon event $\langle addSub|sub \rangle$
  $sub.sender \leftarrow self$
  $connections \leftarrow connections \cup \{(sub.id, connection)\}$
  send $\langle addSub|sub \rangle$ **to** $F_S(sub.id)$

upon event $\langle deleteSub|sub \rangle$
  $connections \leftarrow \{(s,c) \in connections | s \neq sub.id\}$
  send $\langle deleteSub|sub \rangle$ **to** $F_S(sub.id)$

upon event $\langle match|pub, sub \rangle$
  $conn \leftarrow c | (s,c) \in connections | s = sub.id$
  send $\langle match|pub, sub \rangle$ **to** $connection$

State Manager algorithm. A *State Manager* process is responsible for managing the state of a subset of publications and subscriptions active in the cluster. It receives four types of messages from *Frontend* processes: (1) $\langle addPub|pub \rangle$ messages are used to add or update the state of a publication *pub* to the state manager; (2) $\langle deletePub|pub \rangle$ messages are used to delete the state of a publication *pub* from the state manager; (3) $\langle addSub|sub \rangle$ messages are used to add or update the state of a subscription *sub* to the state manager; (4) $\langle deleteSub|sub \rangle$ messages are used to delete the state of a subscription *sub* from the state manager. In addition, a *State Manager* process sends four kinds of message to *Tile Manager* processes: (1) $\langle addTilePub|tile, pub \rangle$ messages are used to add or update the state of a publication *pub* to the *Tile Manager* process responsible for $tile$; (2) $\langle deleteTilePub|tile, pub \rangle$ messages are used to delete the state state of a publication *pub* from the *Tile Manager* process responsible for $tile$; (3) $\langle addTileSub|tile, sub \rangle$ messages are used to add or update the state of a subscription *sub* to the *Tile Manager* process responsible for $tile$; and (4)

Algorithm 2 State Manager

upon event $\langle init \rangle$
  $pubs \leftarrow \varnothing$
  $subs \leftarrow \varnothing$

upon event $\langle addPub|pub \rangle$
  $prev \leftarrow p \in pubs$ **such that** $p.id = pub.id$
  for all $tile \in tiles(prev.Z) \setminus tiles(pub.Z)$ **do**
    send $\langle deletePub|tile, pub \rangle$ **to** $F_T(tile.id)$
  for all $tile \in tiles(pub.Z)$ **do**
    send $\langle addPub|tile, pub \rangle$ **to** $F_T(tile.id)$
  $pubs \leftarrow \{p \in pubs | p.id \neq pub.id\} \cup \{pub\}$

upon event $\langle deletePub|pub \rangle$
  $prev \leftarrow p \in pubs$ **such that** $p.id = pub.id$
  for all $tile \in tiles(prev.Z)$ **do**
    send $\langle deletePub|tile, pub \rangle$ **to** $F_T(tile.id)$
  $pubs \leftarrow \{p \in pubs | p.id \neq pub.id\}$

upon event $\langle addSub|sub \rangle$
  $prev \leftarrow s \in subs$ **such that** $s.id = sub.id$
  for all $tile \in tiles(prev.Z) \setminus tiles(sub.Z)$ **do**
    send $\langle deleteSub|tile, sub \rangle$ **to** $F_T(tile.id)$
  for all $tile \in tiles(sub.Z)$ **do**
    send $\langle addSub|tile, sub \rangle$ **to** $F_T(tile.id)$
  $subs \leftarrow \{s \in subs | s.id \neq sub.id\} \cup \{sub\}$

upon event $\langle deleteSub|sub \rangle$
  $prev \leftarrow s \in subs$ **such that** $id = sub.id$
  for all $tile \in tiles(prev.Z)$ **do**
    send $\langle deleteSub|tile, sub \rangle$ **to** $F_T(tile.id)$
  $subs \leftarrow \{s \in subs | s.id \neq sub.id\}$

Algorithm 3 Tile Manager

upon event $\langle init \rangle$
  $pubs \leftarrow \varnothing$
  $subs \leftarrow \varnothing$

upon event $\langle addPub|tile, pub \rangle$
  for all $s \in subs | s.Z \cap pub.Z \neq \varnothing$ **do**
    if $tile = H_{min}(tiles(pub.Z \cap s.Z))$ **then**
      send $\langle match|pub, s \rangle$ **to** $s.sender$
  $pubs \leftarrow \{(tile, pub)\} \cup$
    $\{(t, p) \in pubs | \neg(t = tile \wedge p.id = pub.id)\}$

upon event $\langle deletePub|tile, pub \rangle$
  $pubs \leftarrow \{(t, p) \in pubs | \neg(t = tile \wedge p.id = pub.id)\}$

upon event $\langle addSub|tile, sub \rangle$
  for all $p \in pubs : sub.Z \cap p \neq \varnothing$ **do**
    if $tile = H_{min}(tiles(p.Z \cap sub.Z))$ **then**
      send $\langle match|p, sub \rangle$ **to** $sub.sender$
  $subs \leftarrow \{(tile, sub)\} \cup$
    $\{(t, s) \in subs | \neg(t = tile \wedge s.id = sub.id)\}$

upon event $\langle deleteSub|tile, sub \rangle$
  $subs \leftarrow \{(t, s) \in subs | \neg(t = tile \wedge s.id = sub.id)\}$

$\langle deleteTileSub|tile, sub \rangle$ messages are used to delete the state state of a subscription *sub* from the *Tile Manager* process responsible for *tile*. Algorithm 2 describes the internals of a *State Manager* processes. When a user registers a new publication or a new subscription, the *Tile Manager* processes that overlap its geographical zone must be notified. In a similar way, when a user updates the geographical zone of a publication or a subscription, some previously notified *Tile Manager* processes must be left and new ones be notified. As we want these actions to be transparent to the end user, a *State Manager* process records the state of publications and subscriptions and sends the necessary maintenance messages across the cluster. When a *State Manager* process receives a message regarding a publication or a subscription, it uses consistent hashing on tile identifiers *tile.id* in order to notify the affected *Tile Manager* processes. Each *State Manager* process is responsible for the states of the publications and subscriptions stored in the *pubs* and in the *subs* sets. As these sets only contain the latest publication and subscription states, their identifiers *pub.id* and *sub.id* are used to identify and eliminate older versions from the sets. We also assume that $tiles(prev.Z)$ returns an empty set when *prev* is null.

Tile Manager algorithm. A *Tile Manager* process is responsible for computing all the matches between publications and subscriptions whose geographical zones overlap with the zone covered by a specific tile. A *Tile Manager* process receives four kind of messages from *State Manager* processes: (1) $\langle addTilePub|tile, pub \rangle$ messages are used to add or update

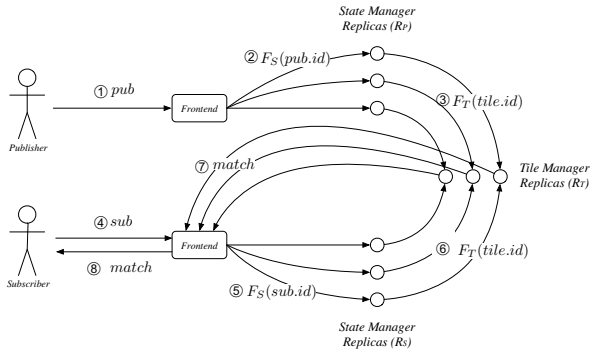


Fig. 6: Replication of a *Match Triggering Graphs*

the state of a publication *pub* in the *Tile Manager* process responsible for *tile*; (2) $\langle deleteTilePub|tile, pub \rangle$ messages are used to delete the state of a publication *pub* from the *Tile Manager* process responsible for *tile*; (3) $\langle addTileSub|tile, sub \rangle$ messages are used to add or update the state of a subscription *sub* in the *Tile Manager* process responsible for *tile*; and (4) $\langle deleteTileSub|tile, sub \rangle$ message are used to delete the state of a subscription *sub* from the *Tile Manager* process responsible for *tile*. In addition, *Tile Manager* processes send match messages in the form of $\langle match|pub, sub \rangle$ to the *Frontend* process attached to the end-user connection of the subscriber. Algorithm 3 shows how collections of publication and subscription states are maintained in *Tile Manager* processes. When a publication state is added or updated, matches are sent to overlapping subscriptions. When a subscription state is added or updated, matches containing the overlapping publications are forwarded to the match filter before reaching the subscription.

E. Fault Tolerance and Reliability

In order to make our architecture reliable and fault tolerant, we adopt an active replication strategy. In other words, each message is processed by all the replicas and, given a replication factor *r*, the system should still be able to compute and deliver all the matches when there are at most $r-1$ failing processes. The general idea behind our replication strategy consists into fully replicating the *Match Triggering Graph*. In order to replicate these graphs, we modify our consistent hashing function so that, instead of returning a single process, it returns a list of replicas that contains the *r* consecutive processes of the hash ring located on distinct physical nodes.

Figure 6 illustrates in more details this replication mechanism. Here, we first notice a different behaviour in the way *Frontend* processes and *State Manager* processes route messages to replicas. In order to duplicate the routing graph, *Frontend* processes contact all the *State Manager* replicas of the lists R_P and R_S. In order to limit the number of messages, a *State Manager* process from the list R_P only sends one messages to a *Tile Manager* process of the list R_T. The position of *State Manager* process in the list R_S is used to select the corresponding *Tile Manager* process in the list R_T. As a result,

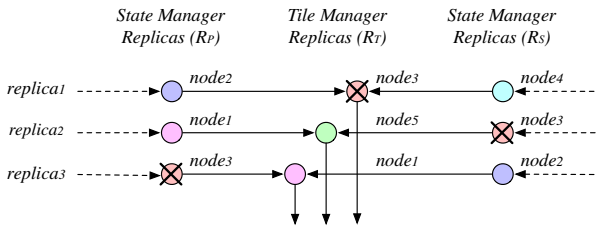


Fig. 7: Non-Independence of failure in the *Match Triggering Graphs*

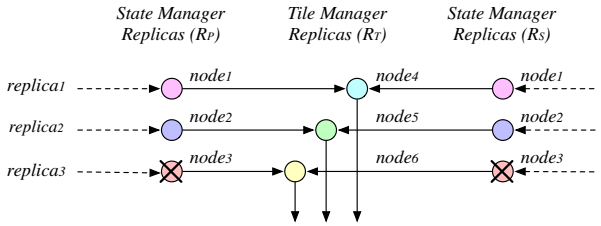


Fig. 8: Ensuring independence of failure in the *Match Triggering Graphs*

the number of messages propagated in the cluster remains proportional to the replication factor.

Independence of failure. The reliability of our architecture relies on the assumption that n node failures do not compromise more than n *Match Triggering Graphs*. However, because of the multiple routing steps involved and the abstraction of virtual nodes typically used in data centers, the lists of nodes associated to the replicated processes can overlap. As a result, several processes along the routing graphs might be located on the same physical node, whose failure might break several graphs and compromise the overall reliability of the system.

Figure 7 illustrates such a failure scenario. Here, if $node_3$ fails, all the replicated routing graphs break. A solution to this problem would be to ensure that the lists of physical nodes associated to the lists of replicas R_P, R_T and R_S never overlap. However, enforcing this property requires some sort of agreement.

Again, our approach consists in finding an a priori agreement that completely avoid the costs of a distributed agreement or the disadvantages of a centralized solution. At the level of the consistent hashing function, we cannot ensure that the selection of replicas throughout the routing steps will result in non-overlapping lists of physical nodes. However, by creating replication groups in the consistent hash ring, we can ensure that the lists of physical machines associated to the selected replicas are either fully overlapping or not overlapping at all. In addition, if we order the lists of replicas by the physical addresses of their hosts, we can guarantee that the overlapping lists will always be aligned, i.e., the replicated processes located on the same machine will communicate with each others. Figure 8 illustrates such a correct alignement. Here the physical nodes associated to the lists R_P, R_T and R_S are either non-overlapping or fully overlapping. In case of full overlapping, the lists are ordered or aligned by using the physical addresses. As a result, the number of broken routing graphs will never be greater than the number of failures, ensuring the reliability of the system.

IV. Theoretical Evaluation

In this section, we theoretically evaluate our distributed algorithm. Given a distributed algorithm, two metrics are generally used to measure its performance: the number of messages required for the termination of an operation and the number of communication steps required for its termination [8].

Regarding the first metric, the operation that has the potential to generate the greatest number of messages in the cluster is the insertion or the update of a publication in the cluster. The number of messages generated by the insertion or the update of a subscription will always be lower because only one subscriber is concerned by the operation. In order to calculate the number of messages required to terminate the insertion of a publication, we first introduce the variable $t = |Tiles(pub.Z)|$ that corresponds to the number of tiles affected by the operation. We also introduce the number of matches m generated by the operation such that $m = |\{sub \in subs | sub.Z \cap pub.Z \neq \varnothing\}|$. On this basis, we can easily enumerate the number of messages generated by the insertion of a publication. To inform the *Frontend* process, a first message is required by the client. A second message is sent by the *Frontend* process to the *State Manager* process. Then, t messages are required to inform the *Tile Manager* processes. As the *Tile Manager* processes are able to agree on which process will send the match message to the *Frontent* process without communicating, m messages will be generated. Finally, m messages (i.e., one message per matching subscriber) are required to forward the matches from the *Frontend* processes to the end user. Hence, we end up with a worst case scenario of $1+1+t+m+m$ messages that correspond to a complexity of $O(t+m)$ messages. As a result, knowing that the worst case of a centralized architecture should be characterized by an $O(m)$ complexity, the overhead introduced by our distributed algorithm is optimal in the considered execution model (presented in Section II).

Regarding the second metric, the number of communication steps required for the termination of the distributed algorithm is bounded to five in the worst case. Although at a first glance this number might seem too high, we demonstrate in the next section that the latency introduced by these steps remains very low, in the context of a data center.

V. Experimental Evaluation

In addition to provide a theoretical evaluation of our algorithms, we tested the scalability of our architecture in the context of a large scale cluster setup. In this section, we describe the results we obtained in terms of throughput, reliability, load, memory and latency.

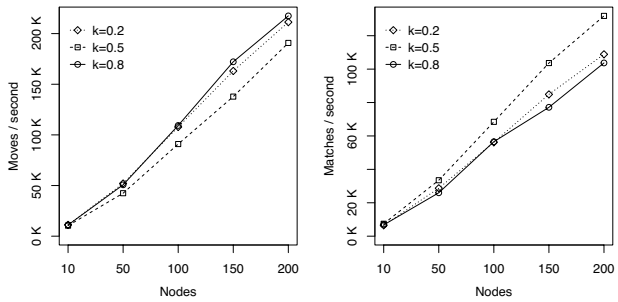


Fig. 9: Varying the ratio of publications and subscriptions (k)

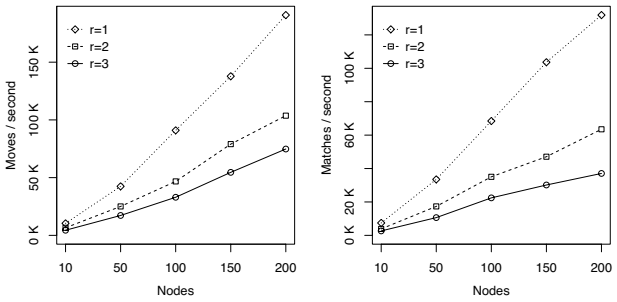


Fig. 10: Varying the replication factor (r)

A. Evaluation setup

Our implementation is written in Scala[6] and rely on a distributed application framework called Akka.[7] This framework relies on the *Actor Model* as an abstraction for distribution, concurrency, parallelism and communication. Therefore, each process described in the architecture corresponds to an actor. We used a Kubernetes[8] cluster hosted in Google Cloud[9] to run our experiments. In Kubernetes, Docker[10] containers are used to accelerate the deployment of applications and a concept named *Daemon Set* can be used to run a copy of an application on a collection of nodes in the cluster. We used StatsD[11] to collect and aggregate statistics across the cluster. We ran our prototype on cluster configurations made of up to 200 virtual machines with two vCPU and 7.5GB of RAM.

B. Cluster settings

In order to verify that our architecture scales horizontally, we saturated various cluster configurations with move operations. To do so, we created a client application that simulates a fleet of moving entities (either publications or subscriptions) of size n. Moving entities have a circular range with radius of 50 meters, a speed of 20km/h, and send move operations every 10 seconds. Each node in the cluster run an instance of this application and the average cpu load of the node is used to mitigate the amount of move operations generated. In other words, the client applications increase the size of the fleet when the load of the nodes are below a minimal threshold. Inversely, the clients decreases the size of the fleet when a maximal load threshold is reached. As a result, the number of move operations is dynamically adjusted to the load that the cluster is able to sustain. The members of the fleet respect a uniform density of 100 moving entities per square kilometres. A configuration parameter k corresponds to the ratio of moving entities mapped to publications. Therefore, $p = k * n$ corresponds to the number of publications and $s = (1-k) * n$ corresponds to the number of subscriptions. In our configurations, the size of the tiles is fixed at a width of approximately 1222 meters. Finally, the parameter r corresponds to the replication factor enforced by the cluster.

[6]https://github.com/scala/scala
[7]https://github.com/akka/akka
[8]https://github.com/kubernetes/kubernetes
[9]https://cloud.google.com
[10]https://github.com/docker/docker
[11]https://github.com/etsy/statsd

C. Horizontal scalability

Figure 9 highlights the scalability of our architecture for three distinct scenarios by varying the k parameter. When $k = 0.2$, we have 20% of publications and 80% of subscriptions, which might correspond to the need of a hailing applications. When $k = 0.5$, the balance between publications and subscriptions is perfect, which might corresponds to the radar of an autonomous fleet of vehicles. When $k = 0.8$, we hypothetically address the need of an IoT application, where the data of many sensors (publications) are aggregated by moving subscriptions. First, in the three cases, we notice that our architecture scales almost linearly. It is interesting to note that, with 200 nodes we were able to sustain the move operations generated by a fleet of 1'900'000 moving entities. Every second, such a load approximately corresponds to 190'000 move operations and 130'000 matches. On a daily basis, this represents a load of more than 15 billion move operations and 11 billion matches. Interestingly, an uneven distribution of publications and subscriptions increases the number of move operations that the system is able to sustain. In fact, as illustrated in Figure 9, such distribution tends to reduce the number of matches and releases additional resources for handling move operations.

D. Reliability overhead

Figure 10 highlights the cost associated to reliability. As illustrated here, replicating the *Match Triggering Graph* symmetrically decreases the number of number of move operations that the system is able to sustain and the number of matches it is able to compute. Interestingly, doubling the replication factor does not divide the throughput by two. This is probably due to the fact that the graph is replicated from the frontend process and not from one end to the other. Finally, it is worth noting that the system scales almost linearly for all the tested replication factors.

E. Load, memory and latency

Given a fixed cluster size of 10 nodes, Figure 11 shows the latency measured in milliseconds over several hours. Here, latency refers to the time taken by a publication to reach

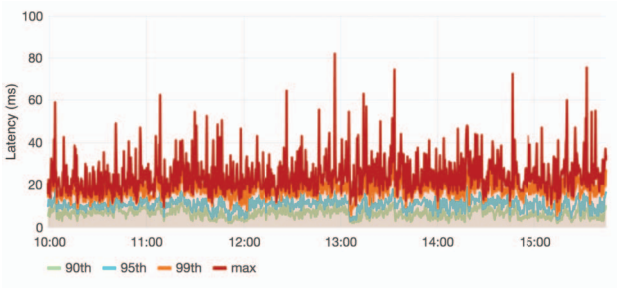


Fig. 11: Latency (milliseconds) on a 10 nodes cluster

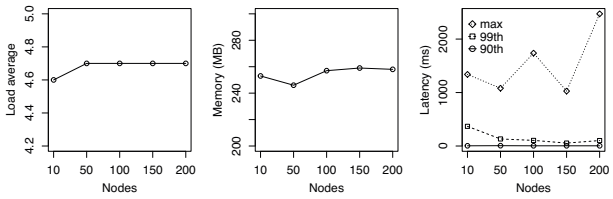


Fig. 12: Load average, memory and latency

a matching subscription. To measure this kind of end-to-end latency, we collocated a subscription and a matching publications on the same nodes and periodically calculated latency by subtracting the sending time of the publication to its reception time. For this experiment, we started virtual machines in two data centers and guaranteed that the virtual machines were located on different hosts. As highlighted here, 99% of the publications were delivered in less than 50 milliseconds and, in the worst case, latency always remained below 100 milliseconds. Interestingly, we noticed that the latency peaks were often linked to stop-the-world garbage collection tasks performed by Java. Average cpu load, memory usage and latency are a key metrics that have to remain stable as the size of the cluster grows. As illustrated in Figures 12, our solution ensures that these metrics remain stable regardless of the scale. In terms of latency, the max value corresponds to the sole maximal latency measure recorded for a given configuration. Here, the 90th and 99th percentile show that most operations are characterized by a very low latency. Therefore, we conclude that the max value is the result of unpredictable garbage collection tasks and not of a greater cluster size.

VI. Related Work

A. Location-Based Publish and Subscribe

To our knowledge, despite the fact that some studies [2], [3], [6], [11] approach location-based publish and subscribe from a distributed-system perspective, none of them focus on horizontal scalability. Interestingly, by addressing scalability in a vertical manner, a huge body of literature recently focused on improving the computation of spatial matches on a single node. For example, Cugola et al. use GPU in order to improve the performance of location-based publish and subscribe [4]. In [21], Yylomenos et al. address the problem of caching data in a publish and subscribe architecture that targets mobile and pervasive devices. In [16] and [22], Guoliang et al. propose an efficient R-Tree based index structure, as well as filtering and pruning algorithms, in order to accelerate the computation of spatial matches. In [12], Hu et al. propose another variant of the R-Tree, the R^I-Tree, to solve the same problem. In [19], Wang et al. propose an alternative index structure, the AP-Tree; it organises records using keywords and spatial data. More recently, in [9], L. Guo et al. described a location-based publish and subscribe system that relies on the concept of safe region in order to reduce communication overhead between a node responsible for the computation of matches and a set of clients. The solution we propose differs from prior work in several ways. For example, in contrast to locality-preserving data structures, such as R-Trees, our solution is somehow counterintuitive because it fragments geographical locality by using consistent hashing in order to improve performance. More importantly, our solution completely departs from approaches that rely on a single node and addresses the problem of horizontally scaling the computation of matches within a cluster.

B. Continuous KNN Queries

Another field of research closely related to location-based publish and subscribe is the processing of continuous top-k nearest-neighbours queries (KNN). Given a query characterised by a moving location and some keywords, the problem consists in continuously updating a set of objects that satisfies the specified constraints in terms of context and content. In [1], Bamba et al. define the notions of spatial alarms and safe-region computation: it relies on the assumption that moving queries are performed on static datasets. Under these conditions, it becomes possible to identify areas in which no matching event will occur. Knowing this, the application can simply disconnect when entering a safe-region and reconnect when leaving it, thus greatly reducing network traffic and communication costs. In [10], Hasan et al. present an efficient construction technique for safe-regions, based on range nearest-neighbor queries. In [20], Wu et al. propose a safe-region detection mechanism that include text relevancy. In [13], Huang et al. introduce an alternative representation for safe-region and improve the performance of safe-region construction. The assumption that a part of the data remains static is unfortunately too strong in our case, since our solution specifically supports moving publications and moving subscriptions.

C. Consistent Hashing

Web services can be subject to what is called the hot spot phenomenon: A web resource can experience a sudden and great popularity, and a single server cannot handle the incoming traffic. In this context, caching and load balancing strategies become vital. In [14], Krager et al. introduced a family of hash functions called consistent hashing: it can be used to relieve hot spots from the web. In [15], the same authors also show how consistent hashing can be used in order

to create distributed caches. Among the four original properties described in these papers, two had a huge impact on distributed database communities. Balance and monotone hash functions give good guarantees on the location and the distribution of cached items. These properties are now at the root of most distributed hash tables; and some popular databases such as Dynamo also use them in order to spread and locate records in a distributed system [5]. Another algorithm, called Rendezvous Hashing [18], developed at the same period achieves the same kind of distributed agreement and can be used as a substitute. Recently, some topic-based publish/subscribe systems use consistent hashing in order to scale horizontally [23], [17], [7]. Contrary to overlapping spatial contexts, topics are strictly isolated from each other. As a result, the architecture we propose for location-based publish/subscribe significantly differs from the usual application of consistent hashing.

VII. Conclusion and future work

We have highlighted the limitations of existing location-based published and subscribe systems in terms of horizontal scalability and have introduced a model adapted to the context of a distributed system. On this basis, we have described and implemented a novel architecture based on the assumption that fragmenting locality by using consistent hashing could help compute matches efficiently in a cluster. Furthermore, we have demonstrated that the number of additional communication steps introduced by our protocol, in order to support moving publications and subscriptions, does not compromise horizontal scalability. In addition, we have shown that reliability and fault tolerance can be achieved with a number of messages proportional to the replication factor. Finally, we have showed with a prototype implementation and an experimental evaluation that our architecture can be deployed on commodity hardware and achieve a very high throughput and preserve a low latency.

Although the proposed distributed architecture yields very promising results, it also also raises a certain number of questions. First, the grid we have introduced in this paper is homogenous, whereas human activity is not. Vast areas, such as oceans are mostly empty and a grid that takes this fact into account might produce interesting results. Consequently, we plan to explore different grid topologies that account for the density of cities and the emptiness of remote areas. Second, the related work regarding safe-region detection highlighted the communication cost linked to continuous transmission of matches. Although safe-regions are based on the assumption that a part of the data remains static, Guo et al. demonstrate in [9] that it can be adapted to moving publications and subscriptions in the context of a single node. Therefore, we plan to investigate the possibility of detecting and constructing safe-regions in the context of a distributed system.

References

[1] B. Bamba, L. Liu, A. Iyengar, and P. S. Yu. Safe region techniques for fast spatial alarm evaluation. Technical report, Georgia Institute of Technology, 2008.

[2] X. Chen, Y. Chen, and F. Rao. An efficient spatial publish/subscribe system for intelligent location-based services. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–6. ACM, 2003.

[3] G. Cugola and J. E. M. de Cote. On introducing location awareness in publish-subscribe middleware. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 377–382. IEEE, 2005.

[4] G. Cugola and A. Margara. High-performance location-aware publish-subscribe on gpus. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 312–331. Springer, 2012.

[5] 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.

[6] P. T. Eugster, B. Garbinato, and A. Holzer. Location-based publish/subscribe. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 279–282. IEEE, 2005.

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

[8] R. Guerraoui and L. Rodrigues. *Introduction to reliable distributed programming*. Springer Science & Business Media, 2006.

[9] L. Guo, D. Zhang, G. Li, K.-L. Tan, and Z. Bao. Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 843–857. ACM, 2015.

[10] M. Hasan, M. A. Cheema, X. Lin, and Y. Zhang. Efficient construction of safe regions for moving knn queries over dynamic datasets. In *International Symposium on Spatial and Temporal Databases*, pages 373–379. Springer, 2009.

[11] A. Holzer, P. Eugster, and B. Garbinato. Alps–adaptive location-based publish/subscribe. *Computer Networks*, 56(12):2949–2962, 2012.

[12] J. Hu, R. Cheng, D. Wu, and B. Jin. Efficient top-k subscription matching for location-aware publish/subscribe. In *International Symposium on Spatial and Temporal Databases*, pages 333–351. Springer, 2015.

[13] W. Huang, G. Li, K.-L. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 932–941. ACM, 2012.

[14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[15] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[16] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 802–810. ACM, 2013.

[17] V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris. Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub. In *Middleware*, 2012.

[18] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking (TON)*, 6(1):1–14, 1998.

[19] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: efficiently support location-aware publish/subscribe. *The VLDB Journal—The International Journal on Very Large Data Bases*, 24(6):823–848, 2015.

[20] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *2011 IEEE 27th International Conference on Data Engineering*, pages 541–552. IEEE, 2011.

[21] G. Xylomenos, X. Vasilakos, C. Tsilopoulos, V. A. Siris, and G. C. Polyzos. Caching and mobility support in a publish-subscribe internet architecture. *IEEE Communications Magazine*, 50(7):52–58, 2012.

[22] M. Yu, G. Li, T. Wang, J. Feng, and Z. Gong. Efficient filtering algorithms for location-aware publish/subscribe. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):950–963, 2015.

[23] Y. Zhao, K. Kim, and N. Venkatasubramanian. Dynatops: a dynamic topic-based publish/subscribe architecture. In *DEBS*, 2013.