Scaling and Load Testing Location-based Publish and Subscribe

Bertil Chapuis, Benoît Garbinato
{firstname.lastname}@unil.ch

Distributed Object Programming Laboratory
University of Lausanne, Switzerland

Abstract—**The rise of the Internet of things (IoT) poses massive scalability issues for location-based services. More particularly, location-aware publish and subscribe services are struggling to scale out the computation of matches between publications and subscriptions that continuously update their location. In this demonstration paper, we propose a novel distributed and horizontally scalable architecture for location-aware publish and subscribe. Our middleware architecture relies on a multi-step routing mechanism based on consistent hashing and range partitioning. To demonstrate its scalability, we present a traffic data generator, which, in contrast to existing generators, can be used to perform real-time load tests. Finally, we show that our architecture can be deployed on a small 10-node cluster and can process up to 80,000 location updates per second producing 25,000 matches per seconds.**

1. Introduction

Today, connected mobility is no longer reserved to valuable living beings. More and more moving objects are connected to the Internet and, with initiatives such as the LoRa Alliance[1], even insignificant objects may soon become talkative. As highlighted by Gartner[2] and IDC[3], this trend is not likely to stop, and there may be between 25 to 30 billion connected objects by 2020.

In this context, when it comes to developing context-aware applications that want to take advantage of the Internet of things (IoT) ecosystem, the location-based publish and subscribe paradigm is of particular interest. With this communication paradigm, connected objects can issue publications and subscriptions that are geographically scoped and that move with them. A match occurs between a given publication and a given subscription if a *context* criterion and a *content* criterion are both met simultaneously. On one hand, a proximity condition between a publication and a subscription can be expressed by a context criterion. On the other hand, a semantical relationship between a publication and a subscription can be embodied by a content criterion.

Although the location-based publish and subscribe paradigm offers a lot of flexibility and expressiveness, its implementation poses difficulties in terms of horizontal scalability. As of today, the systems described in the literature have addressed this scalability issue by proposing *vertically scalable* solutions (i.e., with one computing unit being responsible for the full workload) [1], [2], [3], [4], [5], [6]. Obviously, the exponential growth of the IoT ecosystem can easily exceed the load that is sustainable for a vertically scalable computing unit. In this demonstration, we showcase a horizontally scalable middleware architecture for the location-based publish and subscribe communication paradigm, which can be deployed on clusters made of commodity hardware. In addition, we present near real-time load testing tools that can be used to load test and benchmark such architecture by mocking traffic data in real time.

1. https://www.lora-alliance.org
2. http://www.gartner.com/newsroom/id/3165317
3. https://www.idc.com/getdoc.jsp?containerId=US40755816

2. Middleware Architecture

In this section, we describe a horizontally scalable distributed middleware architecture supporting the location-based publish and subscribe communication paradigm. This architecture addresses the problem of distributing the computation of matches among cluster nodes in a way that allows the overall system to *scale out* or *scale horizontally*. In contrast to centralized spatial data structures, which aim at taking advantage of geographical proximity, our solution uses *consistent hashing* in conjunction with range partitioning expressed with the notion of *tiles* in order to distribute the computation of matches across a cluster.

2.1. Grid and tiles

The notion of grid layout, that divides the world into sets of tiles, is typically used by map services such as Google Maps[4] or Mapbox[5] to serve static data. As illustrated in Figure 1, we use a similar approach to distribute the computation of matches between publications and subscriptions among the nodes of a cluster. However, our use case is more complex since publications and subscriptions continuously move from tile to tile and matches must be emitted in real time in reaction to these updates.

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

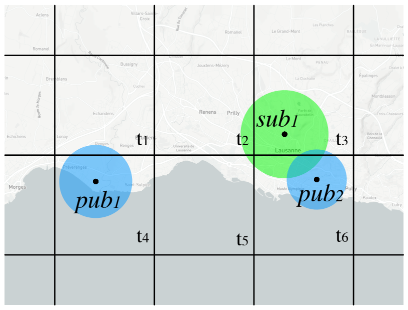


Figure 1: Publications, Subscriptions, and Tiles

2.2. Consistent Hashing

In general, in order to scale horizontally, distributed hash tables partition data items across a cluster of nodes with a family of hash functions called *consistent hashing* [7], [8]. It is common to think about the range of hash values produced by consistent hashing as a ring. In such a ring, the largest possible hash value convolutes to the smallest possible hash value [9]. Each node of the cluster is placed on the ring at a fixed position which can be obtained by hashing the unique identifier of that node. To locate the node responsible for storing a given data item, the identifier of that item is first hashed and then the first node with a placement value greater than the resulting hash value is selected.

2.3. Message Routing

Our architecture relies on range partitions obtained by 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. In other words, consistent hashing functions are responsible for evenly distributing the load on cluster nodes, whereas the subdivision of the earth's surface into tiles is used as a partitioning criteria. Figure 2 shows how these notions of tiling and consistent hashing are assembled together to form our middleware architecture.

Publication Routing. The routing of publications from a moving entity to a tile is illustrated in Figure 2a:

1) When adding a publication to the middleware or updating its location, the moving entity first contacts a frontend service. The frontend service typically runs on every node of the cluster and is placed behind a load balancer, so the moving entity does not know which service is contacted.
2) The frontend service then routes the request to a publication manager service using consistent hashing on the identifier of the publication. This intermediary step is required to manage the changing state of a publication transparently.
3) Finally, publications are routed to a tile manager service using consistent hashing on tile identifiers. Since a publication state changes and can move from tile to tile, some messages are generated to add publications to tiles, while others are generated to remove publications from tiles.

Subscription Routing. As illustrated in Figure 2b, the routing of subscriptions is symmetrical to the routing of publications and achieves the same purpose:

4) The addition or update of a subscription first goes through a frontend service.
5) It is then routed to the subscription manager node responsible for it, using consistent hashing on the subscription identifier.
6) Finally, the subscription is routed to the correct tile manager service using consistent hashing on tile identifiers. As a subscription moves, the subscription manager is responsible for generating the correct tile registration and deregistration messages.

Publication Routing. The tile manager service is responsible for triggering matches when a publication and subscription overlap in a tile. Figure 2c illustrates the routing of a matching publication back to the subscription:

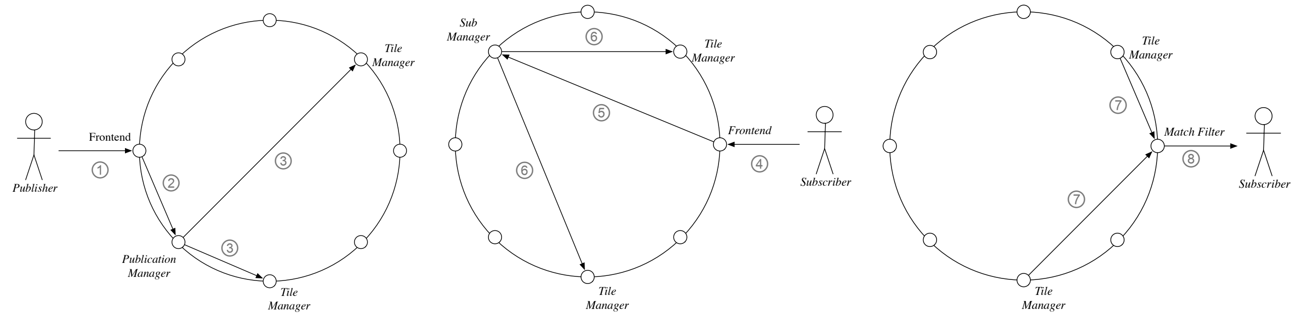
7) Matches between publications and subscriptions are first routed to a match filter service using consistent hashing on the subscription identifier.
8) The match filter may look superfluous but is required since the same match may be computed on several tiles on different nodes. Thus, a filtering mechanism is required and the match filter service is responsible for eliminating duplicates and transmitting matches to subscribers.

3. Traffic Data and Load Testing

Spatio-temporal and traffic data generators are regularly used to benchmark moving object databases. In this section, we highlight some prior batch data generators and show the need for a new kind of generator that produces data in real time with the intent to load test location-based services.

3.1. Batch Generation

The Brinkoff data generator [10] uses a real road network as well as a perturbation model to generate mobility traces. The BerlinMod Traffic Generator [11] relies on the Berlin road network and on the Secondo DBMS to generate data. The Minnesota Traffic Generator [12] provides a web interface to the Brinkoff and BerlinMod traffic data generators. Finally, the Hermoupolis generator [13] uses meaningful semantic data, such as homes and workplaces, to make the generated data more realistic and similar to real human mobility traces. These data generators were devised for use cases characterized by batch processing requirements. They first generate a large dataset, which is then used to benchmark a moving object database. In contrast, our use case requires large volumes of real-time mobility data to load test the scalability of our middleware. Consequently, we devised a real-time data generator that mocks the behavior of a large fleet of moving entities.



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

Figure 2: Distributed routing based on consistent hashing

3.2. Real-time Generation

Our real-time data generator produces synthetic mobility traces, which are distinct from each other and stick to an existing road network. To do so, the underlying generation model assumes that each moving entity performs a round trip that passes by several locations randomly picked in a given range on a map. These round trips are iterative in the sense that, when the round trip of a moving entity ends, the same round trip is started again. To infer the location a moving entity at any given time during a round trip, we rely on an open-source routing library called GraphHopper[6]. GraphHopper uses a variant of the Dijkstra algorithm in conjunction with a road network extracted from the OpenStreetMap[7] dataset to calculate routes and durations between locations. In our case, we use the duration of a route to infer the average speed of a moving entity between two locations of a round trip. In addition, the segments composing the route are used to derive the position of the moving entity at any given time. Finally, since the accuracy of GPS trackers varies and mobility traces never match perfectly to an existing road network, we add some Gaussian noise to the generated traces.

3.3. Load Testing

To load test our middleware, we must be able to simulate many concurrent moving entities. The Scala[8] programming language and the actor model implemented in the Akka[9] suit this requirement perfectly. In terms of implementation, each moving entity corresponds to an actor whose state contains a roundtrip and a position. The generator can be configured with several parameters, such as the number of moving entities, the road network used to produce the data, or the average rate at which moving entities are reporting their locations. Locations are reported by sending publication updates and subscription updates directly to the middleware using a protocol we built using GRPC[10]. In future releases, we might include additional drivers that could be used to test other systems characterized by the same requirements.

6. https://graphhopper.com/
7. https://www.openstreetmap.org
8. http://www.scala-lang.org
9. http://akka.io

4. Demonstration

In this section, we highlight some key characteristics of the interactions proposed to the participants of this demonstration. Through these interactions, our main goal is to show the scalability of the middleware architecture we described earlier.

4.1. Cluster Configuration

The participant interacts with a small cluster made of 10 virtual machines responsible for computing matches between publications and subscriptions. In this cluster, each virtual machine is located on a different host and equipped with two vCPU and 4 GB of RAM. The middleware is deployed using Kubernetes[11] and Docker[12]. Different traffic data generation scenarios are launched with the same tools in the same cluster to demonstrate the scalability of the middleware. For example, with this cluster configuration, our middleware can process up to 80,000 location updates per second for 100,000 moving publications and 100,000 moving subscriptions, producing up to 25,000 matches per second. Interestingly, despite the number of network hops introduced by the architecture, the average end-to-end latency remains below 50 milliseconds.

4.2. Cluster Monitoring

As illustrated in Figure 3, the middleware is monitored with StatsD[13], Graphite[14], and Graphana[15]. The participant interacts with a live dashboard and can obtain real-time

10. http://www.grpc.io
11. https://kubernetes.io
12. https://www.docker.com
13. https://github.com/etsy/statsd
14. https://graphiteapp.org
15. http://grafana.org

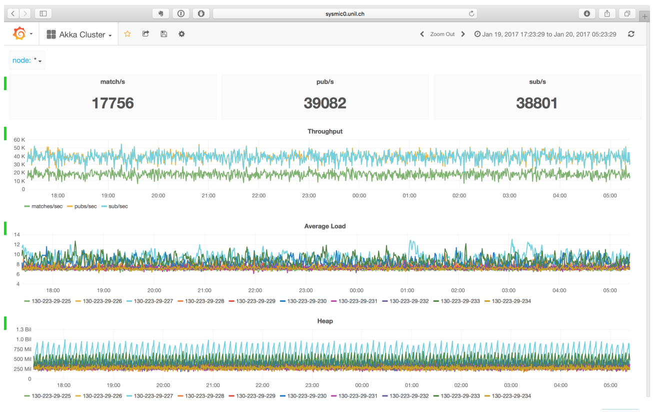


Figure 3: Cluster Monitoring

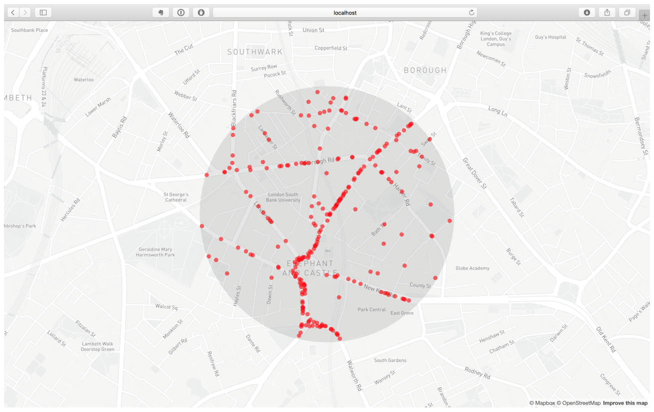


Figure 4: Moving Subscription and Synthetic Publications

insight on the status of the middleware. The displayed metrics include the throughput at which publications and subscriptions are updating their locations, the throughput in terms of matches between publications and subscription and the status of each individual node participating in the computation of matches in terms of load average and memory usage.

4.3. End-User Interactions

As illustrated in Figure 4, a participant can interact with the system with a web browser. Here, the large gray circle corresponds to a subscription that moves as the user navigates the map. The small red circles correspond to the set of moving publications located in the range of the subscription. The publication updates that match the subscription are transmitted to the browser in real time due to the websocket protocol. In this demonstration, the publications update their location every second so the map is animated.

5. Conclusion & Future Work

The middleware for location-based publish and subscribe presented in this demonstration paper achieves horizontal scalability due to a multi-step routing mechanism that relies on consistent hashing and range partitioning. The new traffic data generator we propose can be used to generate large amounts of real-time data to perform load and scalability tests on location-based services. As of today, it is not clear how such a middleware can recover from failure or scale in and out on demand. Consequently, our future work will focus on investigating these issues.

References

[1] 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*. ACM, 2003, pp. 1–6.

[2] 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*. IEEE, 2005, pp. 377–382.

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

[4] A. Holzer, P. Eugster, and B. Garbinato, "Alps–adaptive location-based publish/subscribe," *Computer Networks*, vol. 56, no. 12, pp. 2949–2962, 2012.

[5] 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*. ACM, 2013, pp. 802–810.

[6] 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*, vol. 24, no. 6, pp. 823–848, 2015.

[7] 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*. ACM, 1997, pp. 654–663.

[8] 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*, vol. 31, no. 11, pp. 1203–1213, 1999.

[9] 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*, vol. 41, no. 6, pp. 205–220, 2007.

[10] T. Brinkhoff, "Generating network-based moving objects," in *Scientific and Statistical Database Management, 2000. Proceedings. 12th International Conference on*. IEEE, 2000, pp. 253–255.

[11] C. Düntgen, T. Behr, and R. H. Güting, "Berlinmod: a benchmark for moving object databases," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 18, no. 6, pp. 1335–1368, 2009.

[12] M. F. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel, "Mntg: an extensible web-based traffic generator," in *International Symposium on Spatial and Temporal Databases*. Springer, 2013, pp. 38–55.

[13] N. Pelekis, S. Sideridis, P. Tampakis, and Y. Theodoridis, "Hermoupolis: a semantic trajectory generator in the data science era," *SIGSPATIAL Special*, vol. 7, no. 1, pp. 19–26, 2015.