Event Detection Using Kademlia DHT Network in Distributed Active Data Management Systems

Besim Avci, Saurabh Kadekodi, Arindam Paul

ABSTRACT

Event detection plays a crucial role in active data management, since it is the first component of Event, Condition, Action (ECA) rules. In the case of distributed databases, a responsive fully distributed event detection scheme is yet to be proposed. In this work, we try to achieve a fully distributed reactive event detection mechanism by utilizing a DHT overlay network, namely Kademlia.

1. PROBLEM STATEMENT

In many real world applications, scientific discoveries, and like require collecting data. However, handling big chunks of data is most of the time gruesome. People want to interpret the data to a meaningful subset by defining events of interests, which happen randomly. Examples would be a leakage in a nuclear plant, or a spike in Facebook shares in the stock market. Also, in an event driven architecture or an active database, there is the concept of rules, which consist of three parts;

- *Event:* The signal that triggers the execution of the rule
- *Condition:* The logical test that causes the actoin to be carried out, provided *Event* already occurred.
- *Action:* The part that consists of invocations in the system, like updates.

Basically, action part of a rule executes if condition holds, provided event has already happened. Therefore, detection of events holds a great deal of importance. Since in any architecture that is tightly coupled with events, a robust event detection mechanism is required. To this end, there has been many research attempts to capture an efficient event detection mechanism [7].

First and foremost, an event specification language is necessary to properly define the domain of events, and also to model a system where primitive events can be combined into more complex events. In this paper, we are using a previously defined event specificaiton language, called Snoop [3].

In fully distributed data management systems, which is our relevant use case, every single node is able to detect an event – either externally or internally. In addition to detection of events, every single node is able to subscribe itself to certain events by giving an event definition in the semantics we define. These properties make the system fully distributed with no centralized mechanism. The main challenge of this work is to inform occurrences of events to interested parties with least communication and data overhead.

What separates our work from a publish/subscribe mechanism is that not all event definitions are composed of one simple, primitive event. Events that nodes are interested in will be mostly composite events (events that are constituted by primitive events and event operators) as defined in [3]. This behaviour will make the system much more complicated than a publish/subscribe scheme in the sense that events sources are unknown and there can be multiple sources of events.

2. PRIOR WORK AND PRELIMINERIES

In order to detect, combine, and manage events, a robust expressive event specification language is needed. Thus, we use Snoop semantics [3] for event expressions. However, we limit the event operators to 3; 'and', 'or', 'seq', all of which will be explained later.

Snoop has two main event types; primitive and composite. Primitive events are pre-defined in the system and they are atomic, happen at a time instant and can occur multiple times. Composite events, on the other hand, are the events that constitutes multiple events combined with event operators. Composite events can get as complicated as it gets by definition.

The event operators that can be used in our sys-

tem to make up a composite event are as follows;

1. AND ($\triangle$): Conjunction of two events E_1 and E_2, denoted by $E_1 \triangle E_2$. ($E_1 \triangle E_2$) occurs when both E_1 and E_2 occurs regardless of the order of occurrence.

2. OR ($\bigtriangledown$): Disjunction of two events E_1 and E_2, denoted by $E_1 \bigtriangledown E_2$. ($E_1 \bigtriangledown E_2$) occurs when either E_1 or E_2 occurs.

3. SEQ (;): Sequence of two events E_1 and E_2, denoted by E_1;E_2. (E_1;E_2) occurs when E_2 occurs provided E_1 has already occurred.

Event Represenations.

Every event occurrence is denoted by its event type and event occurrence. For example, the first occurrence of an event type E_1 is represented by e_1^1, second occurrence of the same event type e_1^2. Briefly, subscript represents event type and super-script represents relative time of the occurrence with respect to other occurrences of the same event type.

Parameter Context.

One can take the operators and event occurrences and detect the occurrences of composite events by using boolean algebraic operations. For instance, given the history of events

$$\{\{e_1^1\}, \{e_1^2\}, \{e_2^1\}, \{e_3^1\}, \{e_2^2\}, \{e_4^1\}, \{e_3^2\}, \{e_4^2\}\}$$

and the composite event definition

$$((E_1 \triangle E_2); E3; (E_2 \triangle E_4))$$

using boolean algebra, the composite event occurs 16 times, with occurrences like;

$$\{\{e_1^1, e_2^1, e_3^1, e_2^1, e_4^1\}, \{e_1^1, e_2^1, e_3^1, e_2^1, e_4^2\}, \cdots, \{e_1^2, e_2^2, e_3^2, e_2^2, e_4^2\}\}$$

However, not all 16 occurrences will make sense, and many of them will be unnecessary in most cases. Thus, the authors in [4] propose parameter contexts, which imply different composite event detection mechanisms. They propose 4 different parameter contexts to put composite event occurrences into meaningful shape. For this purpose they name the primitive event that starts the whole composite event as initiator and the one that finishes as terminator. Parameter contexts they introduce are as follows;

1. Recent: The most recent occurrence of the initiator is used for detection. A typical use case would be that of sensor networks, where only the last measurement is needed.

2. Chronicle: For this context, initiator and terminator pair is unique. The oldest initiator and oldest terminator are paired and flushed from the system, very much like applications where there is a causal dependency; for e.g. bug report-release.

3. Continuous: In this context, each initiator starts the detection of the event. This context is useful for trend analysis and forecasting applications where composite event detection along a moving time window needs to be supported.

4. Cumulative: All the constituents events are accumulated for the detection of the composite events, as in banking applications where a customer may withdraw multiple times in a given day, and at the end of the day balance is calculated by accumulating withdraw events.

Hence, different occurrences of the same composite event may happen for different parameter contexts. Note that parameter contexts are tailored for the most common use cases.

Last but not least, a composite event in the network is parsed into subevents and a detection tree is formed to hierarchically detect the composite event by starting from the leaves, which are primitive events, and going upwards in the tree. A sample tree can be seen in Figure 1. Note that parameter contexts need to be applied in every single node of the tree except for the leaves.

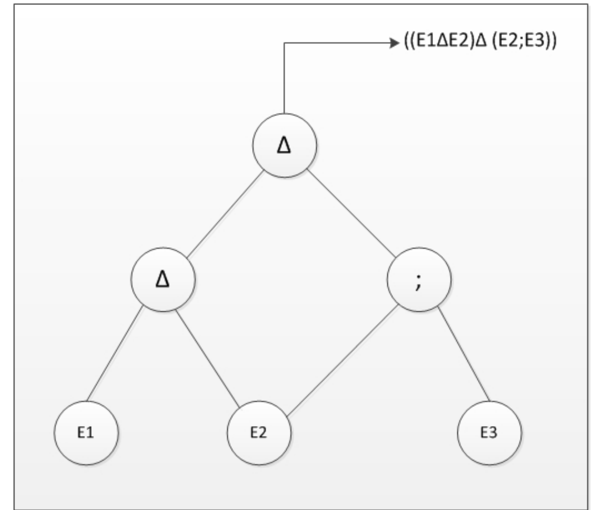

Figure 1: Detection Tree

Moreover, there has been a lot of research interest in publish subscribe systems in the last decade. One of the milestone systems is SCRIBE [2]. Which basically uses a DHT network to create a application level multi-cast messaging capability. Another

set of research attempts that focus on a DHT overlay network are Meghdoot [5] and Willow [8]. These systems let nodes publish events and from a given value-key mapping, data (or message) is sent to the respective DHT nodes and subscribers to these nodes receive data through these nodes, conserving anonymity of both publishers and subscribers. The approach taken in these efforts are same as ours, except for one difference; multiple level of abstraction. To state it differently, we are mapping the composite event detection tree for the whole DHT network, thus any publish event will follow multiple hops in the networks. Benefits of this approach will be discussed in the next section.

3. RESEARCH APPROACH

Our work is noteworthy in that we map the event detection nodes to Kademlia network nodes, which in turn helps us decrease the communication overhead in the network in comparison to conventional publish/subscribe applications. Also, in our system, every node is able to inject an event occurrence and every node is allowed to inject a composite event definition to be notified in case of occurrence.

In addition, we incorporated a time-to-live (TTL) parameter in the system. As the name suggests, a definition injector can define for how long it wants to be notified. Whenever TTL runs out in the system, node automatically unsubscribes itself and all detection tree nodes. Obviously, the algorithm is safe enough to keep the connections for other event detections. Lastly, TTL can be set as forever, which means injector will be notified for occurrences forever.

Main characteristic of our work is that we store event occurrences of an event type in the respective Kademlia node. The way to do so goes by mapping SHA-1 of events to network node ids. Therefore, a node will be responsible for occurrences of event type E_1, and another node will be responsible for event type E_2. Furthermore, all of the event detection tree nodes will be mapped to some node id using SHA-1, which will fall into key space of a network node. Hence, while a network node records occurrences of E_1, another node will be recording, $(E_1 \triangle E_2)$ occurrences for instance.

All primitive events are pre-defined in the system, so whenever a node injects an event, it knows the type of the event but nothing else. Also, a node can inject a composite (can be primitive) event definition and wait to be notified upon occurrence. Whenever a network node injects an event definition, it is parsed in the very same node and a detection tree is formed. Then, every network node, which corresponds to a tree node, is wired to its parent recursively. A simple publish/subscribe mechanism is used in this phase.

Execution Flow

In this section, we will describe how an event detection happens in the network from event definition injection to terminator event occurrence. Event injectors' job is simple: whenever an event occurs, insert it to the network by finding its respective Kademlia node. Event definition is a bit different, though. Event expression along with parameter context is specified, followed by parsing and wiring. An example execution flow is as follows;

1. Definition is inserted and parsed in the node.

2. Detection tree is created and all respective nodes are wired together by subscription mechanism.

3. Primitive event occurred.

4. Event sources notify the respective Kademlia nodes.

5. Event nodes notify whoever is subscribed to them (parent nodes in the trees).

6. Step 5 is repeated until root

7. Event definition injector is notified about the occurrence of the event.

The whole execution flow can be seen in Figure 2.

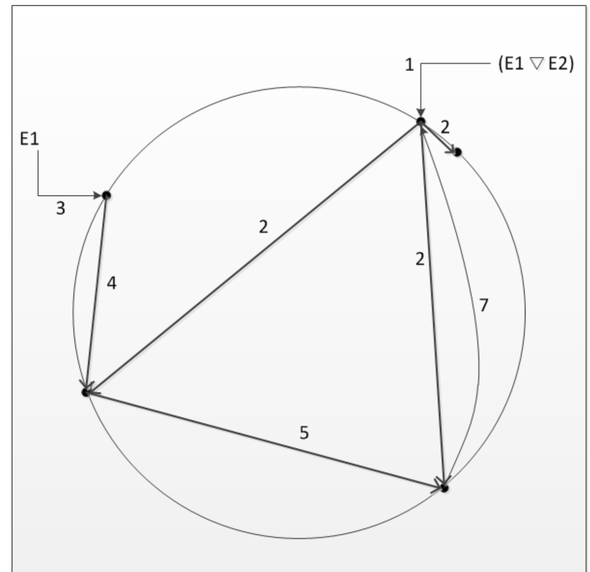

Figure 2: Execution Flow in Key Space

Event Node

The term 'event node' refers to Kademlia nodes which cover the key space where SHA-1 of a particular event falls into. It could be a primitive event or a composite event. To manage the subscriptions in an event node, there is a map structure, where there is subscriber id, parameter context, and a buffer for each subscription. On event occurrence, event node goes through its subscription list and notifies everyone in the list if necessary. Parameter context plays a crucial role in an event node, especially if node is responsible for a composite event, because depending on the parameter context different occurrences may be fired with the same constituent events. Therefore, for each subscriber, event occurrence is calculated based on parameter context and a fire message is sent to the initial subscriber.

4. RESULTS

Our system is running successfully when multiple kademlia nodes are spawned over one physical machine, and also scales well for a small set of machines. At this point in time, we do not put these numbers as we are currently testing only over a small number of machines, a load really unrealistic for a proper working system. Rather, our implementation is more of a proof of concept and unique in its own area. One thing to compare against could be how our project behaves against traditional publish/ subscribe algorithms. Our project will suffer from latency since we make every event occurrence jump over the network for how deep it is in the detection tree. On the other hand, our work saves communication overhead and relieves congestion in the nodes which are responsible for primitive events.

5. LESSONS LEARNED AND FUTURE WORK

This project helped us better understand the usage of a distributed hash table system–yeses and noes. One of the chief assumptions in this project was the resilience of the system. We assumed a system where node failures do not happen. However, replicating data and subscription over couple of nodes and performing a protocol similar to Paxos [6] will enable system to handle node failures. This is the main thing that has been left as a future work. Another future work would be detecting events that already happened before the definition of the composite event. Our current implementation has the foundation for this feature, since we record every occurrence of an event, but we did not implement resilience, so this does not allow us to include history detection feature at the moment. Finally, new approach for event detection has emerged recently –events over time. This new approach assumes that events happen over time, not atomically. As it is explained in the paper [1], it is a simple extension to our proposed approach.

The hardest hurdle we encountered was when a node is notified about occurrence of an event through publish rpc. Since we hold a buffer for each composite event, and there is a different composite event and operator for every subscriber, and every event occurrence can be involved in many composite events, everything gets complicated. If we knew it was going to be this complicated we could have done a better design from the beginning rather than solving the problems as they emerge.

Another future work would be the comparison of this implementation against conventional publish/ subscribe methods. Obviously, conventional publish/ subscribe systems will have lower latency in detection of composite events. On the other hand, publish/subscribe systems will create higher communication overhead and depending on the frequency of event occurrences, it will make some nodes, the ones responsible for primitive events, very congested in terms of network usage. In short, this can be a nice extension to this project since it will lay out how one system is better than the other.

6. SUMMARY AND CONCLUSIONS

Event detection has received attention in active data management context recently, and many research attempts tried to handle event detection component. In this paper, we tried to tackle the problem of event detection in distributed systems where any node can insert events or be interested in certain events. This makes our system completely distributed, and it can be utilized very well in certain use cases, for e.g. sensor networks. Also, using a DHT is one of the most effective solutions to communication in unorganized distributed systems, and it really helped us in handling the communication medium. All in all, this project really made us better enthusiasts for distributed systems.

7. REFERENCES

[1] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, October 2006.

[2] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level

multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC*, 20:2002, 2002.

[3] S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, November 1994.

[4] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[5] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[6] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32, 2001.

[7] David Tam, Reza Azimi, and Hans arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *In International Workshop On Databases, Information Systems and Peer-to-Peer Computing*, pages 138–152, 2003.

[8] Robbert van Renesse and Adrian Bozdog. Willow: Dht, aggregation, and publish/subscribe in one protocol. In *Proceedings of the Third international conference on Peer-to-Peer Systems*, IPTPS'04, pages 173–183. Springer-Verlag, 2004.