University of Trento
Department of Computer Science

An algorithm implementation for Global Predicate Evaluation

Trento, 29/08/2016

Zen Roberto - Bof Michele

Abstract

One core problem in distributed computing is to detect wheter a particular state in a computation can be reached and need to be detected. Problems such as deadlocks detection, monitoring or debugging can be all seen as an instance of the so called Global Predicate Evaluation (GPE) problem. This problem consists in evaluate wheter a given condition called predicate is satisfied among the consistent global states of the system. In order to solve it, one has to face all the practical issues that may arise in a distributed computation: asynchrony and failures of the underlying distributed system, message ordering and inconsistent observations. In this work, we present an implementation that solves GPE based on a simulation of a distributed system. The model built is based on message passing between peers and a monitor which passively observes the system in order to build its global states. At the end of the simulation a given non-stable predicate is evaluated and based on this evaluation graphical results are shown to the user.

1 Introduction

There is a large class of problems that consist in detecting wheter a particular state in a computation can be reached and need to be detected. All of those problems can be considered as subclasses of a well known abstract problem in distributed systems called *Global Predicate Evaluation* (GPE). Its goal is to determine whether the global state of a distributed system satisfies some predicate Φ. Those predicates might be related to the state of the system itself such as identify wheter the system is deadlocked or might be related to the state of the computation that is running among a set of processes. A point that all those subclasses have in common is that all those properties can be encoded in terms of variables. GPE consists in evaluate a predicate over them, so compare the value that those variables have with respect to a given condition. There are two different strategies for solving GPE: one is based on a process called monitor, which actively queries the other processes in order to gather enough information to evaluate the predicate; the other strategy is based on a monitor that passively observes the system and waits for notifications sent by other processes whenever some events are related to the given predicate. Choose among one or the other strategy depends on the underlying distributed system, the communication channels, and other details.

In order to inform the monitor about variable changes, processes have to send messages containing at least information related to the predicate which has to be evaluated. However, this is not the only information they need to carry on. Based on the synchrony or the asynchrony of the model one has to keep into consideration problems that may happen such as message ordering beetween a process and the monitor or between processes themselves. Moreover, messages can also be delayed or lost while processes may fails. Hence, one important component in the system to keep into consideration is the notion of time. Different time measures for events are considered in solving the GPE: global clock, logical clocks and vector clocks. In addition, time is used to collect only observation that are consistent, so observation that could have taken place during an execution.

Consistency of observation is an important property that needs to be satisfied because predicate values are meaningful only when they are evaluated in consistent global states. The monitor has the core role of collect only the consistent global states of the running computation. Based on those states the monitor defines a lattice, an algebraic structure which consists of a partially ordered set (poset) organized in levels with an unique upper bound and an unique lower bound.

1.1 Goals

The reasons which have motivated our group to conduct a study on this topic and the objectives which we have set ourselves are the following:

1. **Simulate a distributed computation**. Structure the project in such a way that users can define how the simulation has to be executed and how long it has to last.

2. **Develop an application as dinamic as possible**. Allow users to define how many peers will run over the system and their behaviour: establish the probability of executing an internal event rather than a message exchange.

3. **Evaluate a Global Predicate among all the consistent global states**. Collect only consistent global states so the monitor can evaluate a given predicate over them.

4. **Display a lattice of the computations**. Display the events executed during the simulation in a lattice structure so users can see all the possible runs that could have been executed and the states satisfying the given predicate.

1.2 Results

We developed a simple solution to the GPE problem based on a monitor that passively observes the distributed computation. The project has been develop in Java using JDK version's 1.7 in combination with the Akka framework which allowed us to build an highly concurrent, distributed, and resilient message-driven application[1]. Our work consists in a simulation of a distributed system in which peers exchange messages and execute internal events while the monitor waits until the end of the simulation. Messages are labelled with a vector clock and contain information related to the predicate that the user wants to evaluate. In particolar, we adopted a simple strategy for structure messages in a way that they were able to transport all the information needed at the very end of the simulation.

Once the simulation ends, the monitor collects the histories of events from all the peers and organize them into a *partially ordered set* (poset) based on the vector clocks recorded in the messages. Once the poset is created, the monitor transform the poset into a lattice that can be seen

[1] `http://akka.io`

graphically by the user. Finally, an analysis of the lattice is conducted: the non-stable predicate given by the user is analized by traversing the lattice (considered as a directed graph) and evaluated among all its possible paths. In this phase we considered either predicates that hold in at least one global state (Definitely) and the ones that hold in all the consistent observations. To conclude, as result we show to the user two pictures that represent lattices of the observed computation with and without the analysis of the predicate.

1.3 Outline

The outline of this report is structured as follows. Section 2 describes the details about the solution we propose, assumptions we posed, the problems we faced and the strategies we adopted to overcome them. In this section we will explain how developed the solution starting from the initialization protocol going through the simulation procedure and ending with the predicate evaluation. We wont go deep into the implementation details, instead we will focus on the logics behind our strategies. The last section states the conclusion of our work and ideas on how we could improve our solution with techniques based on high-level operators that run in parallel.

2 The Solution

We developed an application using the Akka framework which emphasises the concept of *actors* as entities that are independet one each other. In fact, in our project actors can make local decisions, execute events or send messages to other actors. Our solution is based on the fact that a user can specify the duration of the distributed computation and the behaviour of the actors in the system. In our implementation, actors are are both peers and monitor who partecipates in the simulation and interact each other use purely message passing. An important note is that everything in Akka is asynchronous[2].

Altough it is a very simple implementation for solving the GPE problem, different options on how to develop our project were considered. We started developing a real P2P system by running peers for a period of time equal to the one specified by the user. Once this period of time elapsed, the peers send all their history of events to the monitor which evaluates a given predicate on the global states built.

The problem we faced developing this solution was that if the user specifies a period of time long enough, no evaluation is performed until this time elapses. Hence, the user has the costraint of waiting for a period of time equal to the one he specified.

Based on the problem stated above, we decided to refine our implementation and we developed a simulated distributed system. As before, the user specifies the duration of the simulation. However, rather than running peers for a well known period of time, each peer simulates its behaviour during a period of time equal to the one specified by scheduling events in the future. As soon as the last scheduled event is far away in the future from the end of the simulation, the peers simulation ends and the history of events collected is sent to the monitor. In addition, as soon as the monitor collects messages and evaluates a predicate over the consistent global states collected, the execution ends. In this way, we allow users to simulate a long distributed computation without waste his time.

2.1 Assumptions

We decided to work under an asynchronous system, the weakest possible model for a distributed network, because it enables us to estabilish upper bounds on the cost of solving problems in distributed systems[1]. Because we are dealing with this model we have to deal with the problems that might

[2] `http://doc.akka.io/docs/akka/2.4/general/remoting.html`

arise during the execution of distributed computations. In fact, those systems rule out the possibility of processes to maintaining syncronized local clocks so communication remains the only possible mechanism for synchronization in such systems. However, there are no bounds on the relative speeds of processes and there exist no bounds on message delays. Hence, in order to assure that the project works on those systems, we posed the following assumptions for the underlying system and the communication:

1. **Actors are correct**. Both the peers and the monitor are correct: actors cannot be faulty so they cannot fail or behaviour in a bizantine way.

2. **The communication network is strongly connected**. Every peer can communicate directly with every other peer. Each peer knows its neighbours because the actor systems sent a message with all actor which participate to the simulation.

3. **Messages sent are exactly-once delivered**. Based on the Akka framework, messages are at-most-once delivered, meaning that for each message handed to the mechanism, that message is delivered zero or one times (messages may be lost). However, assuming at-most-once delivery implies that we cannot have a consistent sequence of vector clocks. Hence, we need to assume that the channels are reliable such that the messages are always delivered if they were previously sent.

4. **Message ordering is mantained per sender-receiver pair**. For a given pair of actors, messages sent directly from the first to the second will not be received out-of-order. Hence, message buffers between pair of actors are FIFO.

2.2 Initialization

Each simulation starts by executing the **Simulator** class. Firstly, it acquires the configuration specified by the user such as the number of peers in the actual system, the duration of the simulation and more. Secondly, it creates an actor system which generates the peers and the monitor as its children, and then sends a **StartMessage** to them in order to initialize the actors. When the monitor receives the *StartMessage* it starts listening and waiting for **EndMessages** sent by all the peers at the end of their simulation. On the other hand, once a peer received a *StartMessage* it initializes its vector clock and it starts the simulation: it can send messages to other peers.

Although this is a simple protocol in which StartMessages and EndMessages estabilish respectively the start and the end of the simulation, it is not difficult to understand that this procedure may cause problems. In order to explain what may happen, let's describe it in a scenario.

"Suppose that the actors in the system are two peers and the monitor. The actor system sends a StartMessage to all of them but due to the asynchrony of the underlying distributed system, one of the message sent to a peer is delayed. The peer who received the message first initializes its vector clock and start sending messages to the other peer who did not receive yet the StartMessage. If the receipient peer received the message from the peer before the StartMessage sent by the monitor arrived, it cannot update its vector clock because it was not initialized before."

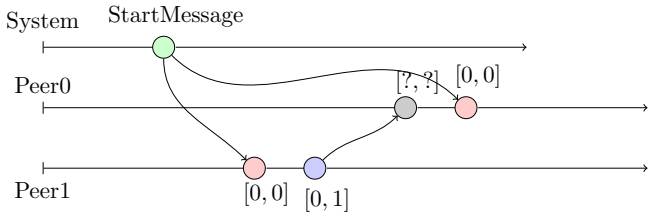

Figure 1: An overcomed issue in our asyncronous simulation.

Figure 1 shows the problem described in the above scenario. StartMessages are sent by the system actor but one of them arrives earlier than the other. Between the two events a message is exchanged and the update of a vector clock is not possible. We overcomed this problem with a simple solution based on the notion of *common knowledge*: "we want the other know that I am ready to start the simulation".

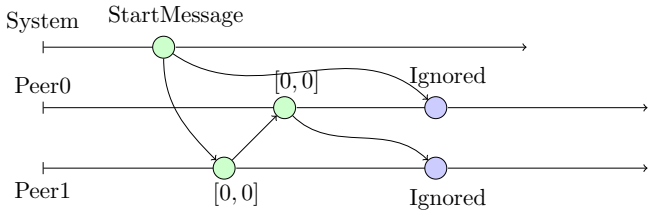

Figure 2: StartMessages exchanged in order to share common knowledge.

Figure 2 displays how we solved the problem related to Figure 1. The first time a StartMessage is received, each peer initializes its vector clock and

sends another StartMessage to everybody else in order to share common knowledge. This procedure is executed only once, the other StartMessages received are ignored. The assumptions made in Section 2 allow us to reason about the correctness of this protocol because message ordering is mantained per sender-receiver pair.

2.3 The simulation

Every time users execute our implementation, a simulation is run. The simulation starts once the initialization phase ends, in other words once each peer in the system has initialized its vector clocks and has sent a StartMessage to every other peer. During the simulation, peers play the main role: they can communicate each other by sending **Messages** or they can decide to execute internal **Events**. In fact, each peer has the possibility either to chose randomly a recipient for a message from the set of neighbours or execute an event based on a given probability defined by the user. However, both the actions are scheduled in the near future, meaning that each peer decides when execute the event based on the remaining time of the simulation. This is possible because the duration of the simultation is decided at priori by the user and the peers know it because it was stored in the *StartMessage* previously received. In order to schedule events, each peer maintains a list of events. Each time the peer schedule a new event, it adds this event into that list. Figure 3 displays the history of events scheduled by a peer.

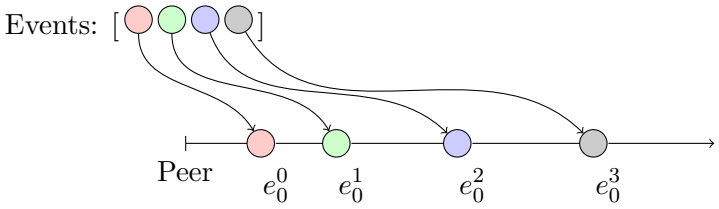

Figure 3: The sequence of events stored in the peer list's and scheduled in the time space diagram.

Each peer is independent from the others, and a possible execution of internal events in combination with messages exchanged between two peers can be seen in the Figure 4. While the first picture on the left represent a sequence of events labbeled with the canonical enumeration, the picture on the right shows the same sequence with events labbeled with their vector

clocks.

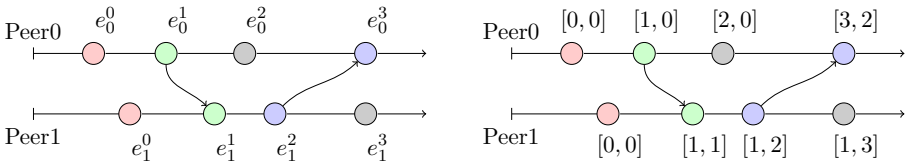

Figure 4: Two time-space diagrams: the one on the left represents a sequence of events labelled using the canonical enumeration, the one on the right instead shows events labelled with their vector clocks.

The scheduling of events is made possible using the Akka API method: `system().scheduler().scheduleOnce()`.

As written in the documentation[3], this method returns an instance of Scheduler which is unique per ActorSystem and is used internally for scheduling things to happen at specific points in time. As can be seen in Figure 5, the simulation is a loop that schedules actions in the future and ends when the next action is scheduled after the end of the simulation. At the very beginning of the simulation, the variable *nextAction* is set to zero, meaning that the simulation is just started. Then, each time the loop is entered, a random time in the future is choosen between *0* and *DELTA_TIME*. As soon as this period of time becomes shorter, the number of events executed during the simulation increases. Once a *nextAction* is scheduled in the future, the peer randomly decides wheter execute an internal event or a message exchange. In both the cases, a peer adds a new event with an updated vector clock to its list of events. However, in the latter case some extra work is needed: a new *Message* is created and sent to a random neighbour. Once the *nextAction* is scheduled after the end of the simulation, the peer sends its list of events to the monitor and stops. As soon as a *Message* is received by a peer, it creates a new event and updates the event with a vector clock based on the vector clock of its previous event and the vector clock stored in the message received. As last step, the peer just adds the new event to the list of events.
As we stated before, events come in two flavors and are described in the **Event** class. Each event has an istance of the **VectorClock** class, an identifier of the peer who executed the event and a variable value which changes along the simulation. Also this variable is related to a probability: users specify the likelihood that a variables change happens. Variables will

[3] `http://doc.akka.io/docs/akka/2.4/java/scheduler.html`

```
while (nextAction < (endTime - startTime)) {

    // Get random integer between 0 and DELTA_TIME.
    // This will be used to schedule an event that will be executed between the next 0 and DELTA_TIME DELTA_TIME_UNIT.
    // Values defined in the Options class.
    rand = new Random();
    nearFuture = rand.nextInt(Options.DELTA_TIME);

    // The time scheduled is defined in term Options.DELTA_TIME and Options.DELTA_TIME_UNIT.
    // It might be the case that this time is defined in Options.SIMULATION_TIME_UNIT, convert it.
    // As written here: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html
    // For example, to convert 10 minutes to milliseconds, use: TimeUnit.MILLISECONDS.convert(10L, TimeUnit.MINUTES)
    if (Options.DELTA_TIME_UNIT != Options.PRECISION_TIME_UNIT) {
        nextAction += Options.PRECISION_TIME_UNIT.convert(nearFuture, Options.DELTA_TIME_UNIT);
    } else {
        nextAction += nearFuture;
    }

    // If the next action is scheduled after the end of the simulation,
    // exit from the loop and send the events history to the monitor.
    if (nextAction >= (endTime - startTime)) {
        break;
    }

    FiniteDuration future = FiniteDuration.create(nearFuture, Options.DELTA_TIME_UNIT);

    // I schedule what to do (send message or execute internal event) in the next future.
    getContext().system().scheduler().scheduleOnce(
            future,
            new Runnable() {
                public void run() {

                    ...
                }
            }, getContext().dispatcher()
    );
}

// After the computation I will send my event history to the monitor
getContext().system().scheduler().scheduleOnce(Duration.create(nextAction, Options.PRECISION_TIME_UNIT),
        monitor, new EndMessage(id, events), getContext().dispatcher(), getSelf());
```

Figure 5: The simulation code.

be used when the predicate will be evaluated. Once the monitor receives *EndMessages* it retrieves the list of events stored in the *EndMessages* and increases the number of messages received. As soon as this number is equal to the number of peers in the simulation the monitor starts collecting global states from the lists of events. The phase regarding the lattice creation and the predicate evaluation is done only if the number of peers is equal to two.

2.4 Lattice creation

In order to evaluate a predicate over the consistent global states, the monitor has to create a lattice of the computations. This is done after receiving the list of events from all the peers. We created this algebraic structure using a directed graph datastructure with particular vertexes defined in the **ProcessVertex** class. These vertexes allow us to store both the index of first peer's event and the index of second's peer event. We defined two similar procedures for building recursively the lattice. The first one just adds the first vertex, what is called the unique supremum while the second

procedure incrementally adds vertex to the structure. Each time, pairs of events from the lists were considered and checked wheter they belong to a pairwise consistent cut via a method defined in the *VectorClock* class. At the end of the procedure, the lattice is a directed graph with only one vertex with no incoming edges and one vertex with no outgoing edges. The lattice is then stored in a file with the *.dot* extension that can be parsed using graphviz command in order to transform it into an image.

2.5 Predicate evaluation

The predicate evaluation is the last phase of our implementation and it is done by traversing the lattice by levels. Depending of the kind of predicate we are evaluating the procedure might change. An important note is that we are considering only non-stable predicates, so the evaluation must be extended to the entire lattice of the computation. As written in [1], there are two choices for defining predicates over computation:

- Possibly(Φ): there exists a consistent observation O of the computation such that Φ holds in a global state of O.
- Definitely(Φ): for every consistent observation O of the computation, there exists a global state of O in which Φ holds.

In our project we allow users to check both options and in case the predicate holds in some states, we store the lattice in a new file and we fill in those states with a color.

3 Conclusion

We developed a simple solution to one of the core problem in distributed system called Global Predicate Evaluation (GPE). Among the two main strategies to solving this problem, we have chosen the one based on a monitor which passively observes the computation among the peers. Peers and monitor are actors who can exchange messages between them and execute internal events. Rather than run a real distributed computation we simulated it using events that were randomly scheduled in the future. This allow us to simulate a long distributed computation without waste users time. Vector clocks have been used as time measure to order the set of events based on the *happens before* relation. Because the system is asynchronous, communication was the only possible mechanism for synchronization and three types of messages were considered: StartMessages, Messages and EndMessages.

References

[1] O. Babaoğlu and K. Marzullo. Distributed systems (2nd ed.). chapter Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, pages 55–96. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.