Epidemics

Antonio Bucchiarone

Fondazione Bruno Kessler, Trento – Italy
bucchiarone@fbk.eu

4 October 2018

Gossip in Distributed Systems

FONDAZIONE BRUNO KESSLER

- It is the *diffusion of information* to a *collection of nodes*.
  - Messages could be broadcast to all nodes.
  - Updates should be applied to all the copies of a replicated database.
  - Chunks of a live streaming video must be delivered to all connected clients.
- Some nodes are aware of a piece of information and some other not.
- The nodes that have it should *cooperate* in order to diffuse these pieces to those that do not.

Problem Definition - I

- We consider a system composed by a *fixed collection P of nodes*, which is known to all nodes (strong assumption).
- Nodes may communicate with each other by *exchanging messages*.
- Nodes may *fail* and *messages can be lost*.
- Each node maintain a single variable *value* whose content is replicated among all nodes.
- Nodes receive updates to this variable:
  - Either by *exchaning messages* between each other, or
  - By means of an *external mechanism* like a *primitive invocation* or a *client request*.
- Updates completely overwrite the content of the variable.
- *Value.time* returns the *timestamp* of the *latest update* known to the node, taken from a totally ordered set of *timestamps T*.

Problem Definition - II

- The ***Goal*** is to spread the updates to all the nodes:
  - If no new updates are injected after some time t, eventually all correct nodes will have the same copy of the variable.
- A node can be in one of three states:
  - **Susceptible (S):** the node does not know about the update;
  - **Infected (I):** the node knows the update and is actively spreading it;
  - **Removed (R):** the node knows the update, but it does not participate in the spreading any more.

- A **susceptible** patient has not contracted the virus (yet).
- An **infected** one is carrying the virus and can infect other patients.
- A patient is **removed** when is not contagious any more.

==> *The goal is to obtain a pandemic*.

SI Model: Anti-entropy

- Three styles of anti-entropy protocols have been proposed:
  - **Push** style
    - Nodes periodically send (*push*) the current content of variable value to a node selected randomly from P.
  - **Pull** style
    - Nodes periodically ask (*pull*) new updates of the replicated variable from other, randomly selected nodes.
  - **Push-Pull** style
    - *Push* and *pull* are combined together.

PUSH Style

1. The *content of the variable is sent* to a *randomly* selected node through a **PUSH message**.

   - Irrespective of whether this node has already received the update or not.

2. When the update is *received*, its timestamp is compared with the local one; if the update is newer, it is copied in value.

PULL Style

1. The *latest timestamp known to p* is sent to a randomly selected peer through a **PULL message**.

2. When a *PULL message is received*, the node compares it with the local timestamp.

   - If the local timestamp is newer, a **REPLY** messages is sent with the content of the value.

   - When a **REPLY** message is received, the timestamp of the received value is compared with the local once again, because other updates may be received in the period between the **PULL** and the **REPLY.**

PUSH-PULL Style

1. The *content of the value* is sent by *p* to a *randomly selected peer q* through a **PUSHPULL** message.

2. When such message is received, the timestamps are compared:
   - If the received timestamp is newer, the update is copied on the local variable.
   - If the received timestamp is older, a **REPLY** message is sent to the sender of **PUSHPULL.**
   - If they are equal, the content is ignored.

Execution Rounds

1. The execution is divided in a set of consecutive rounds of length **Δ** times units.

2. It is represented by the **on timeout** code blocks at the beginning of each version, ended by a **set timeout** operation at the end.

Implementation Assumptions

1. The nodes do not fail.

2. Communication is reliable.

3. Consecutive rounds are synchronized: they start at the same time on all nodes.

4. All the messages sent during the execution of a round (PUSH, PULL, PUSHPULL and their potential REPLY) arrive before the next rounds starts – Round are not intermixed.

ApplicationMain

The *ApplicationMain*, as usual is a place where we can put experiment configurations, which, in this case is only a parameter epiType:

- private static enum EpiType {PUSH, PULL, PUSHPULL};
- private static EpiType epiType = EpiType.PUSH;

Epidemic Actor

- The EpidemicActor is a base class to hide all complicated stuffs.
- 3 other classes that extend this base class: EpidemicPushActor, EpidemicPullActor, and EpidemicPushPullActor.
- In these classes, you need to provide implementation of the 2 following methods for each type of **Anti-Entropy** version):
  - **upon timeout** and
  - **upon receive**

```
protected void onEpidemicTimeoutImpl(); // upon timeout do etc.
protected void onEpidemicReceiveImpl(EpidemicMessage message); // upon receive do etc.
```

Epidemic Value

```
public static class EpidemicValue {
    protected long timestamp = -1;
    protected String value = null;
    public EpidemicValue(long timestamp, String value) {
        this.timestamp = timestamp;
        this.value = value;
    }
    public EpidemicValue(EpidemicValue v) {
        this.value = v.getValue();
        this.timestamp = v.getTimestamp();
    }
    public long getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
    public void copy(EpidemicValue v) {
        this.value = v.getValue();
        this.timestamp = v.getTimestamp();
    }
}
```

Epidemic Message

```
public static class EpidemicMessage {
    protected EpidemicValue value = new EpidemicValue(0, null);

    public EpidemicValue getValue() {
        return value;
    }

    public void setValue(EpidemicValue v) {
        this.value.copy(v);
    }
}
```

Random Process

```
protected ActorRef randomProcess() {
    int index = rand.nextInt(processes.size());
    while (processes.indexOf(getSelf()) == index) {
        index = rand.nextInt(processes.size());
    }
    return processes.get(rand.nextInt(processes.size()));
}
```

OnReceive

```
public void onReceive(Object message) throws Exception {
    if (message instanceof StartMessage) {

        /*
         * Set the peer list
         */
        StartMessage sm = (StartMessage) message;
        processes = sm.group;
        setEpidemicTimeOut();
        runSchedule();
    } else if (message instanceof AssignMessage) {
        AssignMessage am = (AssignMessage) message;
        getValue().setValue(am.getText());
        getValue().setTimestamp(System.currentTimeMillis());
        valueSynced();
    } else if (message instanceof EpidemicMessage) {
        EpidemicMessage em = (EpidemicMessage) message;
        onEpidemicReceive(em);
    } else {
        unhandled(message);
    }
}
```