Master Degree in Computer Science

Concurrency

AA 2014-2015

An Implementation of a non-blocking Binary Search Tree in Java

Project report

Davide Spadini (id 164393)

1 Introduction

A non-blocking Binary Search Tree (BST) is a binary search tree in which the three operations Find(key), Insert(key), and Delete(key) maybe invoked by concurrent threads.

The goal of this project is to implement a BST in Java using the algorithm described in [1]. This algorithm is *non-blocking* (starting from any configuration of any infinite asynchronous execution, with any number of crash failures, some operation always completes) and *linearizable*. The full (and lengthy) proof appears in [2].

2 Usage

Processes are launched virtually together. Identification number *pid* and size of the network n are specified as command line parameters. In order to run a simulation, it is sufficient to launch n instances of the following command on different shells:

```
python consensus.py <pid> <n> [<missing_assumption>]
```

The third parameter is optional and it is used to specify the assumption to be broken. Six choices are given, namely *perfect_channel*, *partially_async*, *not_byzantine*, *max_failures*, *failure_detector* and *fully_connected*. Their meaning and consequences are described below in Sect. 4.
Routing tables for both the node and the failure detector are created automatically, then the oracle is initialized and nodes start to exchange the messages needed by the failure detector protocol. To start the consensus protocol it is sufficient to press *Enter* on every process.

3 Program architecture

Our implementation assumes that the following properties hold:

- **Perfect Channels** [**?**]
  - **Validity - Reliable delivery** A message sent by a correct process will be eventually received by the recipient.
  - **Integrity - No duplication** No message is delivered to a process more than once.
  - **Integrity - No creation** Every delivered message must have been sent by some process.
- **Partially asynchronous channels** The channel is *infinitely often* synchronous.
- **Fully-connected graph** Each process can send messages to every other.
- **Failure by crash** Processes can fail only by crash, excluding byzantine behaviors.
- **Bounded failures** At most $f < n/2$ processes can fail. This bound is imposed by the failure detector protocol we chose.
- **Failure detector** $\diamond S$ There exists an oracle that provides hints about which processes are *operational* or *crashed* and that satisfies the following properties:
  - **Strong Completeness** Eventually *every* faulty process is permanently suspected by *every* correct process.
  - **Eventual Weak Accuracy** Eventually *some* correct process is never suspected by *any* correct process.

The program is divided in two main parts: the failure detector class and the consensus class.

Failure detector The implementation of the failure detector is based on the work of Larrea et al. [**?**], in particular the "Wait-free $\diamond S$ Algorithm".

> The algorithm works in a system in which up to $n-1$ processes can fail (i.e. it is wait free). This algorithm guarantees that all the correct processes eventually converge on the leader process p_{leader} as a common correct process. This property trivially allows the algorithm to provide the eventual-weak accuracy required by $\diamond S$: eventually, p_{leader} will not be suspected by any correct process. The strong completeness property of $\diamond S$ is achieved simply by making every process p_i suspect all processes in the system except p_{leader}.

The pseudo-code of the algorithm is shown in **Algorithm 1**.

Algorithm 1: Failure detector algorithm executed by every process p_i

```
upon init do
    trusted_i ← 1;
    foreach j ∈ {1, ..., i − 1} do
        Δ_{i,j} ← default timeout
// Task 1
every default period do
    if trusted_i = i then
        send(I-AM-THE-LEADER) to p_{i+1}, ..., p_n
// Task 2
upon trusted_i < i and
(not receive(I-AM-THE-LEADER) from p_{trusted_i} in the last Δ_{i,trusted_i} time units) do
    trusted_i ← trusted_i + 1
// Task 3
upon receive(I-AM-THE-LEADER) from p_j and j < trusted_i do
    trusted_i ← j;
    Δ_{i,j} ← Δ_{i,j} + 1
```

The failure detector has a local variable *trusted* which is the guess of the correct process. At the beginning of the execution, it is initialized to 1, meaning that p_1 is the current leader. As the system evolves, $trusted_i$ eventually holds the lowest *id* among those of the processes that have not failed, i.e. the leader.

The leader p_{leader} periodically sends a *I-AM-THE-LEADER* to the processes p_i such that $i > leader$. In fact, p_j can become the leader only if every p_k such that $k < j$ was suspected for failure. p_{leader} is suspected whenever a process p_i does not receive the *I-AM-THE-LEADER* before the timeout $\Delta_{i,leader}$. In that case, $p_{leader+1}$ eventually becomes the new leader.

This algorithm is used by the consensus module as an oracle to guess faulty processes.

Consensus The consensus works in rounds. In every round a node acts as coordinator. A new run is arranged until a decision is made, then a flag *stop* is set. Each process can propose its own value, thus the decision will not be trivial. In *Phase 1* the coordinator broadcasts its proposal, and every participant is blocked until that message is received. Meanwhile, the failure detector is called to check if the current coordinator is suspected for failure. This is a necessary precaution to ensure the termination of the protocol. In *Phase 2* a message is broadcasted by every process. It carries the value previously received, or a *null* data if the coordinator was suspected by the failure detector. Again, the participants have to wait until they have collected enough messages. If all messages carry the same proposal, that value is decided and the decision is broadcasted to let also the slowest node know that the majority has agreed. A flag *decided* is set upon the delivering of the first decision message in order to avoid that a process decides more than once.

Algorithm 2 describes the pseudo-code of the forementioned procedure, taken from [**?**].

Algorithm 2: Consensus algorithm executed by every process p_i

```
upon propose(v_i) do
    r ← 0 ;                                                    // Round
    est ← v_i ;                                                // Estimate
    decided ← false;
    stop ← false;

while not stop do
    c ← r mod n;                                               // Coordinator
    r ← r + 1;

    // Phase 1
    if i = c then
        B-broadcast(⟨PHASE1, r, est, p_i⟩)
    wait B-deliver(⟨PHASE1, r, v, p_c⟩) or p_c ∈ suspects^{fd_i}
    if p_c ∈ suspects^{fd_i} then
        aux ← null
    else
        aux ← v

    // Phase 2
    B-broadcast(⟨PHASE2, r, aux, p_i⟩);
    rec ← ∅;
    proc ← ∅;
    while |proc| ≤ ⌊n/2⌋ do
        wait B-deliver(⟨PHASE2, r, v, p_j⟩);
        rec ← rec ∪ {v};
        proc ← proc ∪ {p_j};
    if rec = {v} then
        est ← v;
        B-broadcast(⟨DECIDE, v⟩);
        stop ← true;
    else if rec = {v, null} then
        est ← v;

upon B-deliver(⟨DECIDE, v⟩) do
    if not decided then
        B-broadcast(⟨DECIDE, v⟩);
        decide(v);
        decided ← true;
```

4 Program evaluation

In the experimental studies, two kinds of test were performed. First of all, we were interested in showing (informally) that the Consensus algorithm actually works, then we loosened one assumption at a time to prove that they are all required to guarantee a correct termination.

Test of correctness To show that the protocol works, we run the algorithm several times using worst-case scenarios and we check if the correct decision was made.

- **Scenario 1** Nodes are slow, but there are no failures. As shown in Table 1, the coordinator is 0 and sends its proposal to the other processes. The failure detectors do not receive the ping of the leader within the delta interval, thus they suspect node 0. Eventually, the ping arrives and they

reset the leader.

Consensus Logs	
Process 0	**Process 1**
Begin of round 0 I'm the coordinator (0), sending PHASE1 Received PHASE1, saving to aux = 0 0 decided for 0	Begin of round 0 Received PHASE1, saving to aux = 0 2 decided for 0
Process 2	**Process 3**
Begin of round 0 Received PHASE1, saving to aux = 0 1 decided for 0	Begin of round 0 Received PHASE1, saving to aux = 0 3 decided for 0
Failure Detectors Logs	
Failure Detector 0	**Failure Detector 1**
INFO:FD 0: task1. trusted is: 0 INFO:FD 0: sending i-am-the-leader to 1 INFO:FD 0: sending i-am-the-leader to 2 INFO:FD 0: sending i-am-the-leader to 3	INFO:FD 1: received ping from 0 INFO:FD 1: task1. trusted is: 0 INFO:FD 1: leader_not_available (delta was 1) INFO:FD 1: Incremented trusted. trusted: 1, timeout: 1 INFO:FD 1: received ping from 0 INFO:FD 1: message received from j(0) < trusted(0). j is the new trusted. trusted: 0, delta: 2 INFO:FD 1: task1. trusted is: 0
Failure Detector 2	**Failure Detector 3**
INFO:FD 2: received ping from 0 INFO:FD 2: task1. trusted is: 0 INFO:FD 2: leader_not_available (delta was 1) INFO:FD 2: received ping from 0 INFO:FD 2: message received from j(0) < trusted(0). j is the new trusted. trusted: 0, delta: 2 INFO:FD 2: task1. trusted is: 0	INFO:FD 3: received ping from 0 INFO:FD 3: task1. trusted is: 0 INFO:FD 3: leader_not_available (delta was 1) INFO:FD 3: received ping from 0 INFO:FD 3: message received from j(0) < trusted(0). j is the new trusted. trusted: 0, delta: 2 INFO:FD 3: task1. trusted is: 0

Table 1: This table describes the scenario 1, in which the failure detectors do not receive the ping from process 0 within the default time interval, so they increment the delta and elect another leader. Eventually the ping from node 0 will be delivered and the leader is reset. After this settlement the nodes will converge to a decision.

- **Scenario 2** Coordinator 0 crashes during *Phase 1*. A decision cannot be made, thus a new round is established. Notice that if the new coordinator 1 had received *Phase 1* message in the previous round, then its estimate would be 0 and the nodes would decide 0. Otherwise, it will propose value 1 and eventually they will decide 1 (as shown in Table 2).

Consensus Logs	
Process 0	**Process 1**
Begin of round 0 I'm the coordinator (0), sending PHASE1 <crashed>	Begin of round 0 Begin of round 1 I'm the coordinator (1), sending PHASE1 Received PHASE1, saving to aux = 1 1 decided for 1
Process 2	**Process 3**
Begin of round 0 Begin of round 1 Received PHASE1, saving to aux = 1 2 decided for 1	Begin of round 0 Begin of round 1 Received PHASE1, saving to aux = 1 3 decided for 1
Failure Detectors Logs	
Failure Detector 0	**Failure Detector 1**
INFO:FD 0: task1. trusted is: 0 INFO:FD 0: sending i-am-the-leader to 1 INFO:FD 0: sending i-am-the-leader to 2 INFO:FD 0: sending i-am-the-leader to 3 <crashed>	INFO:FD 1: received ping from 0 INFO:FD 1: task1. trusted is: 0 INFO:FD 1: leader_not_available (delta was 1) INFO:FD 1: Incremented trusted. trusted: 1, timeout: 1 task1. trusted is: 1 INFO:FD 1: sending i-am-the-leader to 2 INFO:FD 1: sending i-am-the-leader to 3 ...
Failure Detector 2	**Failure Detector 3**
INFO:FD 2: received ping from 0 INFO:FD 2: task1. trusted is: 0 INFO:FD 2: leader_not_available (delta was 1) INFO:FD 2: Incremented trusted. trusted: 1, timeout: 1 INFO:FD 2: received ping from 1 INFO:FD 2: task1. trusted is: 1 ...	INFO:FD 3: received ping from 0 INFO:FD 3: task1. trusted is: 0 INFO:FD 3: leader_not_available (delta was 1) INFO:FD 3: Incremented trusted. trusted: 1, timeout: 1 INFO:FD 3: received ping from 1 INFO:FD 3: task1. trusted is: 1 ...

Table 2: This table describes the scenario 2, in which node 0 crashes right after having sent the *PHASE1* messages. The failure detectors will elect another leader (node 1) and a new round is started. Eventually they will decide for 1.

Loosening the assumptions In the latter case, six situations have been created:

- **Perfect channel** We simulate that process 0 has not perfect channels, specifically it drops all the *PHASE2* and *DECIDE* messages. Under these circumstances, *termination* property will not be satisfied because node 0 will not decide even if it is correct (Table 3).

Consensus Logs	
Process 0	**Process 1**
Begin of round 0 I'm the coordinator (0), sending PHASE1 Received PHASE1, saving to aux = 0	Begin of round 0 Received PHASE1, saving to aux = 0 2 decided for 0
Process 2	**Process 3**
Begin of round 0 Received PHASE1, saving to aux = 0 1 decided for 0	Begin of round 0 Received PHASE1, saving to aux = 0 3 decided for 0
Failure Detectors Logs	
Failure Detector 0	**Failure Detector 1**
INFO:FD 0: task1. trusted is: 0 INFO:FD 0: sending i-am-the-leader to 1 INFO:FD 0: sending i-am-the-leader to 2 INFO:FD 0: sending i-am-the-leader to 3 ...	INFO:FD 1: received ping from 0 INFO:FD 1: task1. trusted is: 0 ... ...
Failure Detector 2	**Failure Detector 3**
INFO:FD 2: received ping from 0 INFO:FD 2: task1. trusted is: 0 ...	INFO:FD 3: received ping from 0 INFO:FD 3: task1. trusted is: 0 ...

Table 3: Process 0 stays blocked in *Phase 2* because it can not receive *PHASE2* or *DECIDE* messages, making it unable to decide.

- **Partially asynchronous channel** An incremental delay is forced on each sending, thus the failure

detector will not be able to distinguish between slow and faulty processes. New rounds keep being started and no decision is reached (Table 4).

Consensus Logs

Process 0

```
Begin of round 0
I'm the coordinator (0), sending PHASE1
Received PHASE1, saving to aux =  0
Begin of round 1
...
```

Process 1

```
Begin of round 0
Received PHASE1, saving to aux =  0
Begin of round 1
...
```

Process 2

```
Begin of round 0
Received PHASE1, saving to aux =  0
Begin of round 1
...
```

Process 3

```
Begin of round 0
Received PHASE1, saving to aux =  0
Begin of round 1
...
```

Failure Detectors Logs

Failure Detector 0

```
INFO:FD 0: task1. trusted is: 0
INFO:FD 0: sending i-am-the-leader to 1
INFO:FD 0: sending i-am-the-leader to 2
INFO:FD 0: sending i-am-the-leader to 3
...
...
...
...
...
...
...
...
```

Failure Detector 1

```
INFO:FD 1: received ping from 0
INFO:FD 1: task1. trusted is: 0
INFO:FD 1: leader_not_available (delta was 1)
INFO:FD 1: Incremented trusted. trusted: 1, timeout: 1
INFO:FD 1: task1. trusted is: 1
INFO:FD 1: sending i-am-the-leader to 2
INFO:FD 1: sending i-am-the-leader to 3
INFO:FD 1: received ping from 0
INFO:FD 1: message received from j(0) < trusted(0).
    j is the new trusted.
trusted: 0, delta: 2
INFO:FD 1: leader_not_available (delta was 2)
...
```

Failure Detector 2

```
INFO:FD 2: received ping from 0
INFO:FD 2: task1. trusted is: 0
INFO:FD 2: leader_not_available (delta was 1)
INFO:FD 2: Incremented trusted. trusted: 1, timeout: 1
INFO:FD 2: received ping from 0
INFO:FD 2: message received from j(0) < trusted(0).
    j is the new trusted.
trusted: 0, delta: 2
INFO:FD 2: leader_not_available (delta was 2)
...
```

Failure Detector 3

```
INFO:FD 3: received ping from 0
INFO:FD 3: task1. trusted is: 0
INFO:FD 3: leader_not_available (delta was 1)
INFO:FD 3: Incremented trusted. trusted: 1, timeout: 1
INFO:FD 3: received ping from 0
INFO:FD 3: message received from j(0) < trusted(0).
    j is the new trusted.
trusted: 0, delta: 2
INFO:FD 3: leader_not_available (delta was 2)
...
```

Table 4: The channels keep delaying ping messages sent by process 0, thus the failure detectors keep increasing the delta. Therefore node 0 continuously switches between suspected and not suspected and new rounds keep being started.

- **Failure only by crash** To simulate a byzantine behavior, we allow process 0 to send different *DECIDE* values to each process, thus they could decide differently (*agreement* property will not be satisfied) (Table 5).

Consensus Logs	
Process 0	**Process 1**
Begin of round 0 I'm the coordinator (0), sending PHASE1 0 decided for 0	Begin of round 0 1 decided for 1
Process 2	**Process 3**
Begin of round 0 2 decided for 2	Begin of round 0 3 decided for 3
Failure Detectors Logs	
Failure Detector 0	**Failure Detector 1**
INFO:FD 0: task1. trusted is: 0 INFO:FD 0: sending i-am-the-leader to 1 INFO:FD 0: sending i-am-the-leader to 2 INFO:FD 0: sending i-am-the-leader to 3 ...	INFO:FD 1: received ping from 0 INFO:FD 1: task1. trusted is: 0 ... ...
Failure Detector 2	**Failure Detector 3**
INFO:FD 2: received ping from 0 INFO:FD 2: task1. trusted is: 0 ...	INFO:FD 3: received ping from 0 INFO:FD 3: task1. trusted is: 0 ...

Table 5: Node 0 sends arbitrary values to the others and everyone will decide differently.

- **Max failures** We allow $n/2 + 1$ processes to fail, thus the correct ones will stay blocked in *Phase 2*. The protocol will not terminate (Table 6).

Consensus Logs	
Process 0	**Process 1**
Begin of round 0 <crashed>	Begin of round 0 <crashed>
Process 2	**Process 3**
Begin of round 0 <crashed>	Begin of round 0
Failure Detectors Logs	
Failure Detector 0	**Failure Detector 1**
INFO:FD 0: task1. trusted is: 0 INFO:FD 0: sending i-am-the-leader to 1 INFO:FD 0: sending i-am-the-leader to 2 INFO:FD 0: sending i-am-the-leader to 3 <crashed>	INFO:FD 1: received ping from 0 INFO:FD 1: task1. trusted is: 0 <crashed>
Failure Detector 2	**Failure Detector 3**
INFO:FD 2: received ping from 0 INFO:FD 2: task1. trusted is: 0 <crashed>	INFO:FD 3: received ping from 0 INFO:FD 3: task1. trusted is: 0 INFO:FD 3: leader_not_available (delta was 1) INFO:FD 3: Incremented trusted. trusted: 1, timeout: 1 INFO:FD 3: task1. trusted is: 1 INFO:FD 3: leader_not_available (delta was 1) INFO:FD 3: Incremented trusted. trusted: 2, timeout: 1 INFO:FD 3: task1. trusted is: 2 INFO:FD 3: leader_not_available (delta was 1) INFO:FD 3: Incremented trusted. trusted: 3, timeout: 1 INFO:FD 3: task1. trusted is: 3 ...

Table 6: The first 3 processes fail so the remaining one can not pass *Phase 2* and it will not decide.

- **Failure detector** We use a dummy failure detector, which never suspects anyone. In our experiments, we crash the coordinator at the beginning of the round, so that the other processes keep waiting for its *PHASE 1* message (Table 7).

Consensus Logs	
Process 0	**Process 1**
`Begin of round 0` `<crashed>`	`Begin of round 0`
Process 2	**Process 3**
`Begin of round 0`	`Begin of round 0`

Table 7: Process 0 crashes and it is not suspected by everyone, therefore they keep waiting for the *PHASE1* message forever.

- **Fully connected network** In this scenario processes are arranged on a ring and the nodes are ordered by their identifier. They can communicate only with their successor. As a consequence, in some cases they cannot collect enough messages and the algorithm will not terminate (Table 8).

Consensus Logs	
Process 0	**Process 1**
`Begin of round 0` `I'm the coordinator (0), sending PHASE1`	`Begin of round 0` `Received PHASE1, saving to aux = 0`
Process 2	**Process 3**
`Begin of round 0`	`Begin of round 0`
Failure Detectors Logs	
Failure Detector 0	**Failure Detector 1**

`INFO:FD 0: task1. trusted is: 0`
`INFO:FD 0: sending i-am-the-leader to 1`
`INFO:FD 0: sending i-am-the-leader to 2`
`INFO:FD 0: sending i-am-the-leader to 3`
`...`
	`INFO:FD 1: received ping from 0`
	`INFO:FD 1: task1. trusted is: 0`
	`...`
	`...`
	`...`

Failure Detector 2	**Failure Detector 3**

`INFO:FD 2: received ping from 0`
`INFO:FD 2: task1. trusted is: 0`
`...`
	`INFO:FD 3: received ping from 0`
	`INFO:FD 3: task1. trusted is: 0`
	`...`

Table 8: A process can send messages only to its successor, thus nobody can collect enough messages to decide.

We want to point out that, in most of the cases, having observed that the processes kept moving from a round to the next one, allowed us to conclude that the algorithm would not terminate.

5 Conclusions and further works

References

[1] Faith Ellen, Panagiota Fatourou, Eric Ruppert, Franck van Breugel *Non-blocking binary search trees* Available at `http://dl.acm.org/citation.cfm?doid=1835698.1835736`

[2] Faith Ellen, Panagiota Fatourou, Eric Ruppert, Franck van Breugel *Non-blocking binary search trees* Technical Report CSE-2010-04, York University, 2010.