Conflict free p2p replicated datatypes

Jim Dowling, Alexandru Ormenisan

April 3, 2017

1 Organization

The course project consists of 4 parts. The first part is simply an introduction to Kompics and is optional if you have worked with Kompics before(it is the same as course ID2203). The rest will be summarised in a final project report which is graded at the end of the course and forms the basis for the project part of the course.

The project is to be done in pairs, with clearly divided responsibilities. It is important to point out the contributions of each member in the final report. Hand in both the report and the source code of your project in Bilda.
Also it is strongly recommended to use git for managing your source code. You are free to use public Github or KTH's gits repository. If you do use either, please provide a link to the repository in your report.

1.1 Goals

The goal of the project is to implement and test peer-to-peer, fully decentralised data types. You are going to implement and test a couple of different versions of CRDT (convergent or commutative data types) that do not require consensus in order to function.

The parts that you will need to implement for this project are:

- Broadcasting through Gossiping
- Basic CRDTs
- Advanced CRDTs

You are free to write your project in either *Java* or *Scala* Kompics.

1.2 Requirements

For this lab you will need the following:

- Java SDK, version 7 or newer. Oracle Java is recommended but OpenJDK should work as well.
- Maven or SBT
- An IDE like Netbeans, Eclipse, IntelliJ is recommended, but any simple text-editor would be enough.

1.3 Grading

Within the project you can get a maximum total of 40 points. Some of the tasks described below cover more than 40 points, allowing you some flexibility in what tasks you want to perform.

- Infrastructure tasks (20p):
  - Task 1.1 - Gossiping Best Effort Broadcast 5p
  - Task 1.2 - Reliable Broadcast 5p
  - Task 1.3 - Causal Broadcast 5p
  - Task 1.4 - Simulations 5p
- CRDTS tasks (one of the following:1,2,3)
  1. Basic 15p
     - Task 2.1 - Sets - G-Set and 2p-Set 5p
     - Task 2.2 - Sets - OR-Set 5p
     - Task 2.3 - Graphs - 2P2P-Graph 5p
  2. Task 2.4 - Loogot - 20p
  3. Task 2.5 - JSON CRDT - 25p

2 Introduction to Kompics

Implement all the PingPong examples from the Kompics tutorial at: `http://kompics.sics.se`.
This task is optional and does not give any points. However, if you haven't worked with Kompics before you should most definitely do it. If you plan on doing the project in Java prioritise the PingPong, if you plan on doing it in Scala, rather to the Programming Exercise first.

3 Skeleton code

You can find the skeleton code at: `https://github.com/Decentrify/id2210-vt17`. Check the repository periodically in case fixes have been released.

4 Infrastructure tasks

In this section you will implement several communication abstraction from the Guerraoui&Rodrigues book [2].

Task 1.1 Gossiping Best Effort Broadcast 5p

In this task you will modify the best effort broadcast abstraction, such that it will still function even without knowing all the peers in the system. You will use a peer sampling system to provide you periodically with random samples from the system. The skeleton code provides you with an implementation of Croupier [1]. In the appendix you can find the the Best Effort Broadcast Properties (figure 1), the Best Effort Broadcast Algorithm (figure 2) and the modified Gossiping Best Effort Algorithm (figure 3).

Task 1.2. Reliable Broadcast 5p

On top of the modified gossiping Best Effort Broadcast abstraction, implement a Reliable Broadcast abstraction. In the appendix you can find the Reliable Broadcast Properties (figure 4) and the Eager Reliable Broadcast Algorithm (figure 5).

Task 1.3. Causal Broadcast 5p

On top of the Reliable Broadcast abstraction, implement a Causal Broadcast abstraction. In the appendix you can find the Causal Broadcast Properties (figure 6) and the No-Waiting Causal Broadcast Algorithm(figure 7).

Task 1.4. Simulation Scenarios 5p

For this task, you will design simulation scenarios that will test the properties of the implemented abstractions. As a minimum, you will design the following simulation scenarios:

- Assume a core of correct nodes - no churn. See that eventually all nodes deliver all broadcast messages
- Assume a core of correct nodes and an extension of nodes that are prone to churn. See that if a correct node sends a message, all correct nodes deliver this message.
- Assume a core of correct nodes and an extension of nodes that are prone to churn. See that if a correct node delivers a message, all correct nodes deliver the message. Design your scenario such that the delivered message comes from a node belonging to the extension and not the core.

In the report describe what properties the scenario try to test. Describe if the properties are safety and/or liveness properties. Do the scenarios give you enough confidence that your implementations do not break these properties?

5 Basic CRDTs

In this section you will implement a couple of CRDT abstractions from Shapiro's CRDT study [4]

5.1 Sets

Sets constitute one of the most basic data structures. Containers, Maps, and Graphs are all based on Sets. We consider mutating operations add (takes its union with an element) and remove (performs a set-minus). Unfortunately, these operations do not commute. Therefore, a Set cannot both be a CRDT and conform to the sequential specification of a set.

Thus, a CRDT can only approximate the sequential set. Hereafter, we will examine a few different approximations that differ mainly by the result of concurrent add(e) and remove(e).

5.2 Task 2.1. G Set and 2pSet 5p

The simplest solution is to avoid remove altogether. A Grow-Only Set (G-Set). The G-Set is useful as a building block for more complex constructions. Implement an operation based version of G-Set. You can only send as messages add operations. Do not send the whole set as message, as you would do in a state based implementation.

The second variant is a Set where an element may be added and removed, but never added again thereafter. This Two-Phase Set (2P-Set). It combines a G-Set for adding with another for removing; the latter is colloquially known as the tombstone set. To avoid anomalies, removing an element is allowed only if the source observes that the element is in the set. Once again, you will implement this abstraction as an operation based version.

For both implementation design a simulation scenario to test your abstraction and describe your findings in the report.

5.3 Task 2.2 OR Set 5p

The Observed-Removed Set (OR-Set), supports adding and removing elements. The outcome of a sequence of adds and removes depends only on its causal history and conforms to the sequential specification of a set. In the case of concurrent add and remove of the same element, add has precedence (in contrast to 2P-Set). The intuition is to tag each added element uniquely, without exposing the unique tags in the interface. When removing an element, all associated unique tags observed at the source replica are removed, and only those. The payload consists of a set of pairs (element, unique-identifier). A lookup extracts element e from the pairs. Operation add(e) generates a unique identifier in the source replica, which is then propagated to downstream replicas, which insert the pair into their payload. Two add(e) generate two unique pairs, but lookup masks the duplicates. When a client calls remove(e) at some source, the set of unique tags associated with e at the source is recorded. Downstream, all such pairs are removed from the local payload. Thus, when remove(e) happens-after any number of add(e), all duplicate pairs are removed, and the element is not in the set any more, as expected intuitively. When add(e) is concurrent with remove(e), the add takes precedence, as the unique tag generated by add cannot be observed by remove.

In the Appendix or the study, you can find the OR Set operation based algorithm (figure 8). For the OR Set implementation design a simulation scenario to test your abstraction and describe your findings in the report.

6 Graphs

A graph is a pair of sets (V,E) (called vertices and edges respectively) such that $E \subseteq VxV$. Any of the Set implementations described above can be used for to V and E. Because of the invariant $E \subseteq VxV$, operations on vertices and edges are not independent. An edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge. What should happen upon concurrent addEdge(u, v) and removeVertex(u)? We see three possibilities:

1. Give precedence to removeVertex(u): all edges to or from u are removed as a side effect. This it is easy to implement, by using tombstones for removed vertices.

2. Give precedence to addEdge(u, v): if either u or v has been removed, it is restored. This semantics is more complex.

3. removeVertex(u) is delayed until all concurrent addEdge operations have executed. This requires synchronisation.

6.1 Task 2.3 2P2P Graph 5p

Using option 1 from above, a 2P2P-Graph is the combination of two 2P-Sets one for vertexes and one for edges. The dependencies between them are resolved by causal delivery. Dependencies between addEdge and removeEdge, and between addVertex and removeVertex are resolved as in 2P-Set.

In the Appendix or the study, you can find the 2P2P Graph operation based algorithm (figure 9). For the 2P2P graph implementation design a simulation scenario to test your abstraction and describe your findings in the report.

6.2 Task 2.4 Logoot 20p

This task replaces tasks 2.1,2.2,2.3. In this task you will study a collaborative editing algorithm with an emphasis on the `undo` operation in the Logoot Undo CRDT algorithm [5].

You will get 5p for doing an analysis on the paper, like you did in the reading assignments and 15p for implementation and simulation scenarios.

6.3 Task 2.5 JSON CRDT 25p

This task replaces tasks 2.1,2.2,2.3. In this task you will study a CRDT that allows arbitrarily embedded lists and maps which can be modified by insertion, deletion and assignment. You will follow the algorithm from the Conflict-Free Replicated JSON Datatype paper [3].

You will get 5p for doing an analysis on the paper, like you did in the reading assignments and 20p for implementation and simulation scenarios.

References

[1] Jim Dowling and Amir H. Payberah. Shuffling with a croupier: Nat-aware peer-sampling. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 102–111, Washington, DC, USA, 2012. IEEE Computer Society.

[2] Rachid Guerraoui and Luis Rodrigues. *Introduction to reliable distributed programming*. Springer Science & Business Media, 2006.

[3] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated json datatype. *arXiv preprint arXiv:1608.03960*, 2016.

[4] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[5] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.

7 Appendix

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: $\langle$ *beb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *beb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

Figure 1: Best Effort Broadcast Properties

Algorithm 3.1: Basic Broadcast

```
Implements:
    BestEffortBroadcast, instance beb.

Uses:
    PerfectPointToPointLinks, instance pl.

upon event ⟨ beb, Broadcast | $m$ ⟩ do
    forall $q \in \Pi$ do
        trigger ⟨ pl, Send | $q, m$ ⟩;

upon event ⟨ pl, Deliver | $p$, $m$ ⟩ do
    trigger ⟨ beb, Deliver | $p$, $m$ ⟩;
```

Figure 2: Best Effort Broadcast Algorithm

Algorithm 1 Gossiping Best Effort Broadcast

Implements:

Gossiping Best Effort Broadcast, **instance** *gbeb*.

Uses:

PerfectPointToPointLink, **instance** *pp2p*.

BasicSample, **instance** *bs*.

```
 1: upon event ⟨ $Init$ ⟩ do
 2:     $past := \emptyset$
 3: upon event ⟨ $gbeb, Broadcast \mid m$ ⟩ do
 4:     $past := past \cup (self, m)$
 5: upon event ⟨ $bs, Sample \mid s$ ⟩ do
 6:     for all $p \in s$ do
 7:         trigger ⟨ $pp2p, Send \mid p,$ [HistoryRequest] ⟩
 8: upon event ⟨ $pp2p, Deliver \mid p,$ [HistoryRequest] ⟩ do
 9:     trigger ⟨ $pp2p, Send \mid p,$ [HistoryResponse, $past$] ⟩
10: upon event ⟨ $pp2p, Deliver \mid p,$ [HistoryResponse, $history$] ⟩ do
11:     $unseen := history \setminus past$
12:     for all $(pp, m) \in unseen$ do
13:         trigger ⟨ $gbeb, Deliver \mid pp, m$ ⟩
14:     $past := past \cup unseen$
```

Figure 3: Gossiping Best Effort Broadcast Algorithm

Module 3.2: Interface and properties of (regular) reliable broadcast

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: $\langle$ *rb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *rb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Figure 4: Reliable Broadcast Properties

Algorithm 3.3: Eager Reliable Broadcast

```
Implements:
    ReliableBroadcast, instance rb.

Uses:
    BestEffortBroadcast, instance beb.

upon event ⟨ rb, Init ⟩ do
    delivered := $\emptyset$;

upon event ⟨ rb, Broadcast | $m$ ⟩ do
    trigger ⟨ beb, Broadcast | [DATA, self, $m$] ⟩;

upon event ⟨ beb, Deliver | $p$, [DATA, $s$, $m$] ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup \{m\}$;
        trigger ⟨ rb, Deliver | $s$, $m$ ⟩;
        trigger ⟨ beb, Broadcast | [DATA, $s$, $m$] ⟩;
```

Figure 5: Eager Reliable Broadcast Algorithm

Module 3.9: Interface and properties of causal-order (reliable) broadcast

Module:

Name: CausalOrderReliableBroadcast, **instance** *crb*.

Events:

Request: $\langle$ *crb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *crb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

CRB1–CRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

Figure 6: Causal Broadcast Properties

Algorithm 3.13: No-Waiting Causal Broadcast

```
Implements:
    CausalOrderReliableBroadcast, instance crb.

Uses:
    ReliableBroadcast, instance rb.

upon event ⟨ crb, Init ⟩ do
    delivered := $\emptyset$;
    past := [];

upon event ⟨ crb, Broadcast | $m$ ⟩ do
    trigger ⟨ rb, Broadcast | [DATA, past, $m$] ⟩;
    append(past, (self, $m$));

upon event ⟨ rb, Deliver | $p$, [DATA, mpast, $m$] ⟩ do
    if $m \notin$ delivered then
        forall $(s, n) \in$ mpast do                          // by the order in the list
            if $n \notin$ delivered then
                trigger ⟨ crb, Deliver | $s, n$ ⟩;
                delivered := delivered $\cup \{n\}$;
                if $(s, n) \notin$ past then
                    append(past, $(s, n)$);
        trigger ⟨ crb, Deliver | $p$, $m$ ⟩;
        delivered := delivered $\cup \{m\}$;
        if $(p, m) \notin$ past then
            append(past, $(p, m)$);
```

Figure 7: No-Waiting Causal Broadcast Algorithm

Specification 15 Op-based Observed-Remove Set (OR-Set)

```
1:  payload set $S$                                   ▷ set of pairs { (element $e$, unique-tag $u$), ... }
2:      initial $\varnothing$
3:  query $lookup$ (element $e$) : boolean b
4:      let $b = (\exists u : (e,u) \in S)$
5:  update $add$ (element $e$)
6:      atSource $(e)$
7:          let $\alpha = unique()$                   ▷ $unique()$ returns a unique value
8:      downstream $(e, \alpha)$
9:          $S := S \cup \{(e, \alpha)\}$
10: update $remove$ (element $e$)
11:     atSource $(e)$
12:         pre $lookup(e)$
13:         let $R = \{(e,u) | \exists u : (e,u) \in S\}$
14:     downstream $(R)$
15:         pre $\forall (e,u) \in R : add(e,u)$ has been delivered  ▷ U-Set precondition; causal order suffices
16:         $S := S \setminus R$                      ▷ Downstream: remove pairs observed at source
```

Figure 8: OR Set - operation based

Specification 16 2P2P-Graph (op-based)

```
1:  payload set $VA$, $VR$, $EA$, $ER$
2:                                                    ▷ $V$: vertices; $E$: edges; $A$: added; $R$: removed
3:      initial $\varnothing, \varnothing, \varnothing, \varnothing$
4:  query $lookup$ (vertex $v$) : boolean b
5:      let $b = (v \in (VA \setminus VR))$
6:  query $lookup$ (edge $(u,v)$) : boolean b
7:      let $b = (lookup(u) \wedge lookup(v) \wedge (u,v) \in (EA \setminus ER))$
8:  update $addVertex$ (vertex $w$)
9:      atSource $(w)$
10:     downstream $(w)$
11:         $VA := VA \cup \{w\}$
12: update $addEdge$ (vertex $u$, vertex $v$)
13:     atSource $(u,v)$
14:         pre $lookup(u) \wedge lookup(v)$          ▷ Graph precondition: $E \subseteq V \times V$
15:     downstream $(u,v)$
16:         $EA := EA \cup \{(u,v)\}$
17: update $removeVertex$ (vertex $w$)
18:     atSource $(w)$
19:         pre $lookup(w)$                           ▷ 2P-Set precondition
20:         pre $\forall (u,v) \in (EA \setminus ER) : u \neq w \wedge v \neq w$   ▷ Graph precondition: $E \subseteq V \times V$
21:     downstream $(w)$
22:         pre $addVertex(w)$ delivered              ▷ 2P-Set precondition
23:         $VR := VR \cup \{w\}$
24: update $removeEdge$ (edge $(u,v)$)
25:     atSource $((u,v))$
26:         pre $lookup((u,v))$                       ▷ 2P-Set precondition
27:     downstream $(u,v)$
28:         pre $addEdge(u,v)$ delivered              ▷ 2P-Set precondition
29:         $ER := ER \cup \{(u,v)\}$
```

Figure 9: 2P2P Graph - operation based