Distributing Thread-Safety for Reactive Programming

In-Progress Paper

Drechsler, Joscha
Technische Universität Darmstadt
Darmstadt, Hessen, Germany
drechsler@cs.tu-darmstadt.de

Mezini, Mira
Technische Universität Darmstadt
Darmstadt, Hessen, Germany
mezini@cs.tu-darmstadt.de

Abstract

Reactive Programming (RP) is a programming paradigm for implementing interactive applications modularly and declaratively. Many interactive applications today are distributed, and thus supporting RP for distributed applications is an interesting research avenue. One of the key benefits of RP is that it avoids "callback hell" through providing consistent update semantics in the form of glitch-free change propagation. Previous research has developed separate solutions for decentralized glitch-free change propagation and thread-safe glitch-free change propagation. This paper aims to combine these works and provide an algorithm for glitch-free change propagation that is both thread-safe *and* decentralized, and thus suitable for distributed applications.

1 Introduction

Reactive Programming (RP) [1] is a programming paradigm for implementing interactive applications modularly and declaratively. It provides two fundamental programming abstractions, referred to as reactives: Events for event-based programming, and Signals for constraint-based state management similar to spreadsheet formulae. Signals and Events are combined and derived from each other, and these dependency relations are tracked by the run-time to form the directed acyclic dependency graph of an application.

Imperative processes in applications (e.g., user pressing a key) can change designated input reactives. RP then propagates these changes across the dependency graph, recomputing all (transitively) affected reactives. RP systems use propagation algorithms, which assess the topology of the dependency graph to provide glitch freedom consistency during this change propagation. Glitch freedom ensures a well-defined update order for reactives' recomputations and thus avoids "callback hell", where applications are subjected to obfuscated, incidental control flow. It ensures that invariants (e.g., an index Signal is never out of bounds of a corresponding array Signal) of programs during quiescence (i.e., while no changes are propagating) also hold while dependent reactives are being recomputed. Glitch freedom thus is a major contributor to the ease-of-use of RP.

REBLS'18, August, 4, 2018, Boston, MA, USA
2018.

To facilitate the implementation of distributed applications with RP, locally defined Events and Signals can be shared remotely [4, 11, 12]. Other hosts can use remotely received reactives transparently as if they were local: They can derive new local reactives from them, or combine them with other local or remotely received reactives. In distributed RP, the dependency graph thus is a distributed data structure, with some dependency edges crossing network edges.

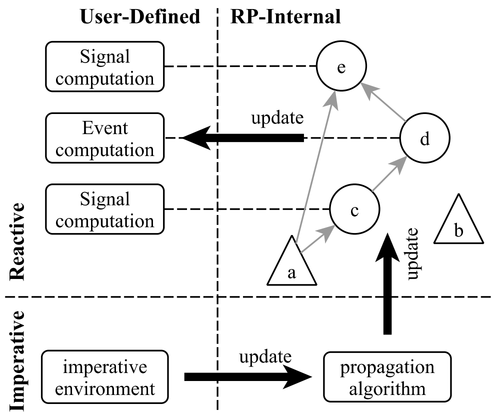

Figure 1. Anatomy of local, single-threaded RP

Figure 1 shows the components of a local, single-threaded reactive application, and how they call each other to propagate changes: The imperative environment (bottom left) submits changes to input reactives (triangles) to the propagation algorithm (bottom right), which instructs dependent reactives (circles) in the dependency graph (top right) to recompute by (re-)evaluating their defining user computations (top left). The classically used propagation algorithm in this setting achieves glitch freedom by processing all affected reactives through a priority queue. In the distributed setting, this is not feasible because the queue would become a centralized coordinator that must sequentially process all affected reactives of the distributed dependency graph.

Several recent works propose decentralized propagation algorithms [7–10], but most lack support for at least one of RP's features (Signals, Events, or dynamic changes of the dependency graph topology). The SID-UP propagation algorithm in Distributed REScala [4] is the only algorithm to support all of these features, but can not actually be applied

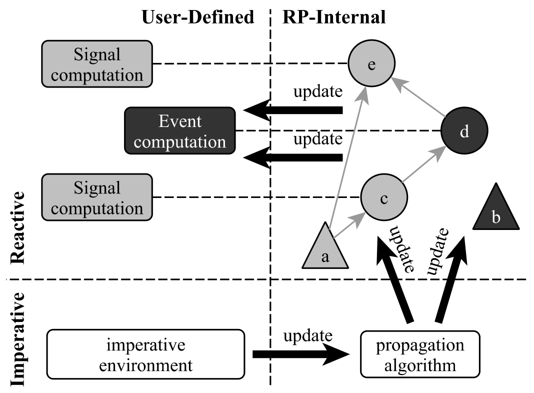

Figure 2. Anatomy of distributed, single-threaded RP

to distributed applications without adding a centralized coordinator to prohibit concurrency. Figure 2 visualizes, how SID-UP models a distributed reactive application: Reactives in the dependency graph – along with their defining computations – exist on different hosts, visualized through light and dark gray shades. Beyond that though, SID-UP assumes that only one single-threaded imperative thread exists in the entire application (a 1:n relation between threads and hosts), which in practice is almost never true. In addition to the dependency graph becoming a distributed data structure, distribution implies that multiple CPUs collaborate across network connections, which means that every distributed system is naturally also concurrent with multi-threading. Multiple propagation algorithm instances may execute simultaneously, leading to race conditions between their operations that break any consistency guarantees if not controlled. Thus, a distributed propagation algorithm must be not only decentralized, but also thread-safe.

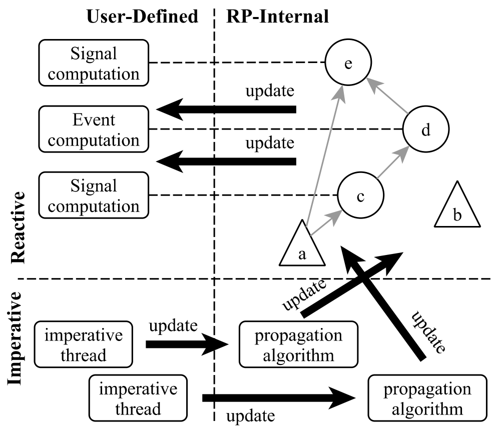

Figure 3. Anatomy of local, multi-threaded RP

Thread-safety for Reactive Programming has been solved only very recently, and only for the local setting. MV-RP [2] is a scheduler add-on for propagation algorithms to add thread-safety on top of any glitch-free change propagation. Figure 3 visualizes the anatomy of local, but multi-threaded RP applications as supported by MV-RP: Multiple imperative threads spawn multiple concurrently executing instances of propagation algorithms, and their operations are subject to race conditions when concurrently accessing the single dependency graph in shared memory (a n:1 relation between threads and hosts). MV-RP intercepts the method calls between imperative environment, propagation algorithm and dependency graph. Calls by the imperative environment are implicitly wrapped into transactions and used to precompute update schedules on the dependency graph. Subsequent calls to the dependency graph, once the propagation transaction is executing, are then internally ordered, delayed and executed according to these schedules. Overall, this orchestration results in *serializable* execution for transactions: they execute concurrently, but all values visible to user code look identical as if transactions executed sequentially, one after the other. Moreover, by exploiting synergies with RP semantics, MV-RP never needs to abort any transactions due to, e.g., being stuck in deadlocks, even under dynamic changes of the dependency graph topology.

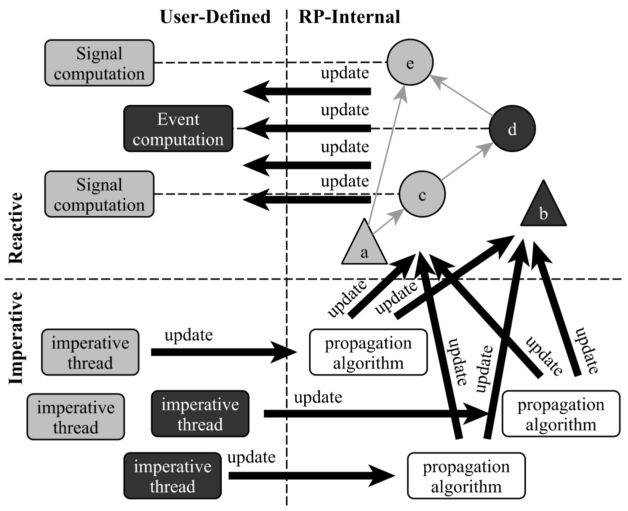

Figure 4. Anatomy of distributed RP

Unfortunately, it is not feasible to simply add the MV-RP scheduler for thread-safety on top of the SID-UP distributed propagation algorithm. Figure 4 visualizes the anatomy of distributed RP applications with concurrency; we again use light and dark gray shading to visualize different hosts: Multiple threads on multiple hosts spawn multiple propagation transactions, all of which can affect reactives on remote hosts (a n:n relation between threads and hosts) where they can interact with any transactions from any remote hosts. The MV-RP scheduler globally orders interacting transactions in the directed acyclic *stored serialization graph*, which in this setting thus also becomes a distributed data structure that all hosts try to query and modify concurrently. To ensure linearizability for modifications of the stored serialization graph, the original (local-only) MV-RP uses a singleton exclusive lock, which is unsuitable for distributed applications. This shows that not only the propagation algorithm must be decentralized, but also the concurrency scheduler.

In this paper, we devise a decentralized variation of MV-RP, to integrate with the SID-UP decentralized propagation algorithm. To this end, Section 2 presents a detailed analysis of the implications of distribution on all data structures of MV-RP and all their operations, and derives their requirements for the distributed setting. Section 3 then engineers a prototype decentralized implementation of the MV-RP scheduler. Finally, Section 4 discusses the combination of SID-UP and MV-RP, presenting redundancies and synergies between the algorithms that allow for simplifications and optimizations.

2 The Impact of Distribution on MV-RP

MV-RP uses two data structures: node version histories and the stored serialization graph (SSG). We investigate both.

2.1 Node Version Histories

Every reactive holds one instance of a node version history. Each node version history is a list of versions – planned and past values of the reactive's variables. These include references to all reactives that the reactive depends on (predecessors), or that depend upon the reactive (successors). Thus, node version histories are the backbone of the dependency graph, i.e., form a distributed data structure. Node version histories provide the following linearizable operations:

- **now, after/depend, before**: read the user value in various temporal relations to the current execution context.
- **reev-in**: start reevaluating after all dependencies have become glitch-free.
- **reev-out**: complete reevaluating, add/remove predecessor edges for new/obsolete dependencies, and retrieve the current set of successors (since they may have become ready to reevaluate).
- **drop, discover**: add/remove a reactive to/from the set of successors.

Each of these operations only affects variables within the respective node version history instance. Therefore distributing node version histories does not require any operations to be implemented linearizable across remote references. Node version histories may store remote references and pass them around through parameters and return values of their operations, but they never need to operate over them. Therefore, node version histories do not have any algorithmic requirements in order to be used in a distributed setting.

The only aspect worthy of consideration is that all values of remotely shared reactives will usually be read (operations of the first bullet point) on every remote host at least once, and often even multiple times, by dependant reactives and regular processes. As such it is advisable to cache replicas of remotely shared reactives' values on all remote copies, to avoid redundant network traffic and minimize the latency of such reads. This replication is not a big challenge though. MV-RP stores transactions values separately, and each transaction writes each reactive's only once and never changes it again. This means, writing reactives' values is an append-only process and therefore easy to replicate because replicated values never become outdated and never require invalidation. Moreover, new values can be pushed out piggybacked onto the messages to re-check dependant reactives on remote hosts for glitch-free reevaluation readiness. This way, eager replication of reactives' values can be implemented at almost no extra cost.

2.2 Stored Serialization Graph

In the directed acyclic SSG, MV-RP tracks the partial serialization order of all transactions, i.e., the order in which they appear to atomically execute. It is constructed incrementally, with new ordering relations being established when transaction execute operations on reactives where others have previously executed non-commutative operations. Transactions in the SSG store their current phase (Framing while precomputing an update schedule, then Executing, and finally Completed), and a set of predecessor and/or successor transactions. Since all hosts in a distributed application should be allowed to start transactions, predecessor and successor relations may involve remote references, meaning the SSG is also a distributed data structure. We list all operations supported by the SSG and how they are affected by distribution:

- **Exists Path:** Query, in which direction two transactions are ordered, or if they are not ordered yet. MV-RP uses "Exists Path" to determine valid placement of transactions in node version histories. As such, it is executed very often (insertion into each reactive's sorted history of n versions requires $log_2(n)$ queries), particularly more often than new edges are created. Further, it is accessed heavily concurrently, by multiple concurrent reevaluations within multiple concurrent transactions. Thus, "Exists Path" requires a fast and highly concurrent implementation, meaning ideally non-blocking and backed by – in distributed applications – locally replicated data. Fortunately, like reactive's values, SSG replication is not challenging, because the SSG is also an append-only data structure: Only new edges are added, but existing ones are never changed and never become invalid.
- **Add Acyclic Edge:** If a transaction is not ordered against all other transactions in a node version history that it accesses, MV-RP optimistically decides on an order where it should be placed. It then tries to implement this order globally, by establishing up to one predecessor and successor edge through "Add Acyclic Edge" for the directly preceding or succeeding transaction in that history (all other relations then follow by transitivity). The order may (have) become redundant or invalid though, due to other

threads also establishing ordering relations concurrently. To guard against such race conditions, "Add Acyclic Edge" atomically executes the following steps: First, use "Exists Path" in the reverse direction of the edge to check if the edge would close a cycle. Second, insert the edge only if it does not create a cycle. As an extreme example, two reactives, independent of each other in the dependency graph topology, may simultaneously request the insertion of two edges between four different transactions, but other previously established relations between these transactions transitively imply that inserting both these edges would close a cycle. In a distributed application, both reactives, all transactions and all previously established relations involved in this scenario may originate from different hosts. This operation therefore poses very challenging algorithmic requirements, making it very difficult to implement decentralized. It requires to verify the absence of a potentially distributed path and to prevent concurrent executions on any other hosts from creating such a path until the insertion of the respective edge has been completed and communicated to these hosts.

- **Iterate Predecessors:** MV-RP allows transaction to progress their phase from Framing to Executing or Executing to Completed only after all predecessor transactions have already done the same progression. Therefore, the SSG must provide a way for transactions to iterate their current predecessors. The data already required for "Exists Path" queries is sufficient to iterate all predecessors. Thus this operation does not introduce any further requirements in distributed applications.
- **Get Phase:** Transactions must be able to query the phase of other transactions for three reasons. First, to ensure that phase switches occur only after predecessors' (see previous point). Second, when creating transaction orders, framing transaction must never ordered before executing transactions, to maintain the invariants established by the first point. Third, garbage collection needs to know if transactions are completed, in order to remove no longer needed versions and SSG nodes and edges. Again, the phase of a transaction is likely to be read much more often than written (at most two phase switches). Thus this data should also be replicated in distributed applications. Phase progression is also append-only though (transaction only ever switch to the next phase, but never back), and thus this replication again is not very challenging.
- **Progress Phase:** Transactions progressing their phase may allow some successor transactions to progress their own phase next. In distributed applications, such successor transactions may reside on remote hosts. Thus, a distributed implementation must support notifying successor transactions through remote messages. Because phase progression is append-only, this does not require any kind of synchronization. Moreover, if current phases are replicated on remote hosts, these notifications can be piggybacked onto the phase replica updates. Thus "Progress Phase" also introduces no further requirements and can be implemented at almost no extra cost.

To conclude, distributing the SSG is straight forward in that all operations except "Add Acyclic Edge" require at most replicating append-only data. Only "Add Acyclic Edge" has complex algorithm requirements, for which finding a decentralized implementation is very challenging because it requires synchronization across an arbitrary number of independent hosts. In the following section, we present one prototype solution to this problem.

3 Building Blocks for Distributed MV-RP

"Exists Path" and "Add Acyclic Edge" in the original local-only MV-RP are based on a partially dynamic graph algorithm for maintaining the transitive closure under only edge insertions, but not deletions [6] – the precise scenario that MV-RP requires. In its original form, this algorithm implements operations "Get Path" and "Add Edge" both in O(n) amortized time, by internally maintaining for each node one hash-indexed spanning tree of transitively reachable nodes. MV-RP adapts this algorithm to implement "Exists Path" and "Add Acyclic Edge" in the following ways.

The original "Get Path" uses the source's hash index to find the sink's node in the source's spanning tree in O(1). If the spanning tree node does not exist in the hash index, then there is no path from source to sink. If it does exist, traversing all nodes along the spanning tree upwards to the root returns one proven path from sink to source in O(n). MV-RP uses only the first step of this implementation – looking up the spanning tree node in O(1) – and then immediately returns true or false if it exists or not, which implements "Exists Path" in O(1). "Add Acyclic Edge" is implemented as a conditional "Exists Path" query with sink and source reversed, to ensure that no reverse transitive path exists, and then "Add edge". This puts "Add Acyclic Edge" in O(1 + n), i.e., also in O(n).

Linearizability for "Exists Path" is based on the internally maintained spanning trees being grow-only data structures: "Add Edge" may insert additional nodes into the hash index and spanning trees, but all existing elements remain unchanged. By using a thread-safe non-blocking hash index for the spanning tree nodes, the "Exists Edge" O(1) query thus automatically becomes a linearizable and fully non-blocking operation. This implementation can simply be reused in a distributed implementation where the spanning tree index is replicated to each remote host.

Linearizability for "Add Acyclic Edge" on the other hand is achieved by mutual exclusion through a central lock for two reasons. First, it ensures that there are no racing writes from concurrent "Add Edge" executions that could corrupt the hash-indexed spanning trees. Second, it ensures that the preceding "Exists Path" query to prevent cycles returns a reliable result in that no racing "Add Edge" can establish this path concurrently. Because of the centralized lock, this implementation is not feasible to use in distributed applications, and a decentralized implementation is necessary instead. The next section presents a modified union-find data structure that we call lock-union-find, that allows us to associate one lock with each connected component of an undirected graph. The section afterwards then discusses a decentralized "Add Acyclic Edge" implementation based on lock-union-find.

3.1 Lock-Union-Find

A union-find data structure [5] associates a single representative element for each connected component of a graph under the partially dynamic case of edge additions, but not removals. Each node maintains a pointer that is either null, indicating the representative of a connected component, or points to a different node, in which case the representative can be reached by following these pointers recursively – this is the "Find" operation. When an edge is inserted between two nodes, the representatives of both nodes' connected components are looked up through "Find", and then one's pointer is set to the other, unless they are the same already. This way, the nodes of both connected components are unified under the same representative – the "Union" operation.

```
procedure lock(node):
  let found := find(node)
  if (CAS(found.ptr, null, found)) return found
  else return null

procedure lockUnion(node1, node2):
  let locked := lock(node1)
  if (locked == null) return null
  else:
    let found := find(node2)
    if (found == locked) return locked
    else if (CAS(found.ptr, null, locked)):
      return locked
    else:
      locked.ptr := null // unlock
      return null
```

Figure 5. Pseudocode of "Lock" and "LockUnion".

Our lock-union-find data structure extends union-find twofold: First, it adds a third pointer state: A node whose pointer points to itself represents that the connected component is currently locked. Second, it adds a "Lock" operation, and replaces "Union" with "LockUnion". Figure 5 shows pseudocode implementations[1] for both operations:

- **"Lock"** can either succeed or fail. It executes "Find" and then tries to atomically compare-and-set the representatives pointer from null to the representative itself. If that succeeds, "Lock" succeeds and returns the now-locked representative. If it failes, "Lock" fails and returns null.
- **"LockUnion"** can also either succeed or fail. It first tries "Lock" on the first node. If that fails, "LockUnion" also fails. Otherwise, it continues with "Find" on the other node. If the second node's found representative is the same as the first node's locked representative, "LockUnion" succeeds immediately, since both nodes are already part of the same now-locked connected component. Otherwise, "LockUnion" next tries to atomically compare-and-set the found representative's pointer from null to the locked representative. If that succeeds, "LockUnion" succeeds with both nodes now unified within the same locked connected component. Otherwise, the locked representative is unlocked again (pointer reset to null) and "LockUnion" fails.

[1]In practice, both implementations are significantly more complex since they perform additional tasks that are not relevant here, e.g., non-blocking path compression (for near O(1) amortized runtime complexity) and non-blocking manual reference counting (for distributed garbage collection).

Semantically, both "Lock" and "LockUnion" fail if they involve any already locked connected component. Lock-union-find thus associates an exclusive lock with each connected component of a graph that, while held, also prevents the connected component from growing. The exclusive use of atomic compare-and-set instructions ensures linearizable execution for both "Lock" and "LockUnion" without further coordination, meaning their implementations are decentralized. By allowing remote references in the pointer, lock-union-find thus is feasible to use in distributed applications.

```
procedure addAcyclicEdge(source, sink):
  if(existsPath(source, sink)) return true
  if(existsPath(sink, source)) return false
  let locked = lockUnion(source, sink)
  if (locked == null): // tail recursive
    return addAcyclicEdge(source, sink)
  else try {
    if(existsPath(source, sink)) return true
    else if(existsPath(sink, source)) return false
    else:
      let changes = addEdge(sink, source)
      updateAllReplica(changes)
      return true
  } finally { locked.ptr := null } // unlock
```

Figure 6. Pseudocode of "Add Acyclic Edge".

3.2 Mutual Exclusion through Lock-Union-Find

Figure 6 shows the pseudocode implementation of the distributed decentralized "Add Acyclic Edge" implementation. It adds a lock-union-find layer over the SSG of replicated hash-indexed spanning trees of transactions. Each newly created transaction initially forms a connected component by itself. To order two transactions, "Add Acyclic Edge" first repeatedly tries to "LockUnion" them. If an order is established by racing concurrent threads, "Add Acyclic Edge" terminates prematurely. Otherwise, "LockUnion" eventually succeeds with both transactions unified in a single locked connected component. Then, "Add Edge" is executed locally, the resulting changes are pushed to all replica of the affected hash-indexed spanning trees, and finally the connected component is unlocked again.

Mutual exclusion per connected component is more fine-grained locking than what local-only MV-RP implemented, allowing concurrent executions of "Add Acyclic Edge" in some cases. We show that "Add Acyclic Edge" is still linearizable though, and thus the correctness of MV-RP is unaffected by this change: If concurrent "Add Acyclic Edge" operations would create a cycle (or interact with each other in any other way), all involved transactions would afterwards be part of the same connected component. Therefore, checking "Exists Path" and executing "Add Edge" only after the respective transactions have been unified into one exclusively locked connected component prevents this case.

4 Synergizing MV-RP and SID-UP

We now consider the combination of decentralized MV-RP with decentralized SID-UP [4]. In its essence, SID-UP is a single-pass variation of a two-pass parallel mark-and-sweep (mark pass: mark all affected nodes dirty, sweep pass: re-validate any dirty nodes once they have no more dirty predecessors). It uses source identifier sets to determine the results of the mark pass implicitly during the sweep pass, and thus works without executing mark passes: Every node holds a set of all sources it is reachable from. Any given node would have been marked by a preceding mark pass, if the intersection of its source set and the set of sources from which a given change propagation originated is non-empty.

SID-UP's basic implementation exerts a lot of redundant effort, because each node for every received re-validation message has to iterate all its predecessors and determine if any are still marked. SID-UP thus developed several optimizations to its single-pass sweep phase [3]:

- Nodes with only a single static dependency always reevaluate immediately upon re-validation messages. They never need to assess all their predecessors, because they can never have a second dependency that is not re-validated.
- All other nodes only iterate their predecessors a single time, instead of once for each received re-validation message. Before processing the first re-validation message of each change propagation, each node iterates its predecessors to count the number of marked ones. This value is used to initialize a "marked predecessors" counter. A second "changed predecessors" counter is initialized with 0. Each re-validation message carries a "changed" bit that reflects, whether or not the sending node changed its value. For each received re-validation message, the "marked predecessors" counter is decremented, and – if the "changed" bit is set – the "changed predecessors" counter is incremented. Once the "marked predecessors" counter reaches 0, all predecessors have been re-validated and the node can re-validate. If the "changed predecessors" counter is not 0, the at least one predecessor has a new value and the node must thus recompute its own value to re-validate itself. Both counters are further affected by dynamic changes of the dependency graph's edges: If an edge is added (removed) from the dependency graph, then on the edge's sink node (a) the "marked predecessors" counter is incremented (decremented) if the edge's source node is marked, and (b) the "changed predecessors" counter is incremented (decremented) if the edge's source node changed its value previously within the same change propagation.

Opposing SID-UP's single-pass philosophy, MV-RP has shown that strong consistency under multi-threading requires a two-pass algorithm in order to avoid deadlocks between concurrent change propagations. This means, that any extra effort exerted by propagation algorithms to provide glitch freedom in only a single pass change propagation (e.g., SID-UP maintaining source identifier sets on each node) is redundant, assuming equivalent results can be achieved by piggybacking a mark pass on MV-RP's Framing phase. In the case of SID-UP, not only is this possible, but the concurrency scheduling provides the required information for free already, and no extra effort needs to piggybacked on. For any change propagation, the set of marked nodes for which SID-UP would compute a non-empty source set intersection is identical to the set of nodes on which MV-RP allocates placeholder versions while pre-scheduling updates in the Framing phase. The Executing phase's sweep pass thus can simply consider every node as marked iff it has a placeholder version. This way, the combination of SID-UP and MV-RP works without source identifier sets.

A further optimization becomes possible due to the combination of two-pass mark-and-sweep with SID-UP's counters optimization. Instead of nodes initializing their "marked predecessors" by iterating all predecessors before processing the first re-validation message of each propagation, all "marked predecessors" can be counted incrementally during the framing phase. Whenever a transaction's Framing phase reaches a node, the "marked predecessors" counter of a newly created placeholder is initialized with value 1, or the counter of the already existing placeholder is incremented. This way, when the framing phase completes, every node's "marked predecessors" counter has already been set up correctly. Thus, with this combination of SID-UP plus MV-RP, nodes never need to iterate their predecessors, meaning this extended optimization also supersedes SID-UP's other operation.

As a final aspect, MV-RP requires propagation algorithms to implement an additional interface for (de)queueing additional reevaluations when dynamic changes of dependency edges execute with a mismatch between serializability time and real time. In the combination with optimized SID-UP, the implementation of this interface is exactly the regular implementation of SID-UP's counter updates on dynamic dependency changes. The only difference is, that after a single transaction added or removed a single dependency edge, MV-RP will execute the respective counter updates for that transaction *and* all later transactions.

5 Conclusion and Future Work

We have analyzed the requirements for a distributed RP propagation algorithm for glitch-free change propagation that is both decentralized and thread-safe, based on a combination of the SID-UP propagation algorithm and the MV-RP concurrency scheduler. We have presented a decentralized variant of the MV-RP scheduler, and discussed its combination with SID-UP, pointing out redundancies and synergies which result in both simplification and optimization. In the future, we want to design an evaluation for distributed RP, to either show the usability of our solution, or discover its weak spots and research better implementation alternatives.

Acknowledgments

This work has been supported by the European Research Council, grant No. 321217.

References

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4, Article 52 (2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[2] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-Safe Reactive Programming. In *Proceedings of the 2018 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '18)*. ACM, New York, NY, USA.

[3] Joscha Drechsler and Guido Salvaneschi. 2014. Optimizing Distributed REScala *(REBLS'14)*.

[4] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming *(OOPSLA '14)*. ACM. https://doi.org/10.1145/2660193.2660240

[5] Bernard A. Galler and Michael J. Fisher. 1964. An Improved Equivalence Algorithm. *Commun. ACM* 7, 5 (May 1964), 301–303. https://doi.org/10.1145/364099.364331

[6] G.F. Italiano. 1986. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science* 48 (1986), 273 – 281. https://doi.org/10.1016/0304-3975(86)90098-8

[7] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (July 2018), 689–711. https://doi.org/10.1109/TSE.2018.2833109

[8] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

[9] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I Now Pronounce You Reactive and Consistent: Handling Distributed and Replicated State in Reactive Programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2016)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3001929.3001930

[10] José Proença and Carlos Baquero. 2017. *Quality-Aware Reactive Programming for the Internet of Things.* Springer International Publishing, Cham, 180–195. https://doi.org/10.1007/978-3-319-68972-2_12

[11] Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web *(Onward! 2014)*. ACM, New York, NY, USA, 55–68. https://doi.org/10.1145/2661136.2661140

[12] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. In *Proceedings of the 2018 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '18)*. ACM, New York, NY, USA.