Guaranteeing Correctness and Availability in P2P Range Indices

Prakash Linga, Adina Crainiceanu, Johannes Gehrke, Jayavel Shanmugasudaram
Cornell University, Ithaca, NY

linga,adina,johannes,jai@cs.cornell.edu

Abstract

New and emerging P2P applications require sophisticated range query capability and also have strict requirements on query correctness, system availability and item availability. While there has been recent work on developing new P2P range indices, none of these indices guarantee correctness and availability. In this paper, we develop new techniques that can provably guarantee the correctness and availability of P2P range indices. We develop our techniques in the context of a general P2P indexing framework that can be instantiated with most P2P index structures from the literature. As a specific instantiation, we implement P-Ring, an existing P2P range index, and show how it can be extended to guarantee correctness and availability. We quantitatively evaluate our techniques using a real distributed implementation.

1 Introduction

Peer-to-peer (P2P) systems have emerged as a promising paradigm for structuring large-scale distributed systems. The main advantages of P2P systems are scalability, fault-tolerance, and ability to reorganize in the face of dynamic changes to the system. A key component of a P2P system is a P2P index. A P2P index allows applications to store (value, item) pairs, and to search for relevant items by specifying a predicate on the value. Different applications have different requirements for a P2P index. We can characterize the index requirements of most P2P applications along the following three axes:

- **Expressiveness of predicates**: whether simple equality predicates suffice in a P2P index, or whether more complex predicates such as range predicates are required.
- **Query correctness**: whether it is crucial that the P2P index return all and only the data items that satisfy the predicate.
- **System and Item Availability**: whether it is crucial that the availability of the P2P index and the items stored in the index, are not reduced due to the reorganization of peers.

For example, simple file sharing applications only require support for equality predicates (to lookup a file by name), and do not have strict correctness and availability requirements (it is not catastrophic if a search occasionally misses a file, or if files are occasionally lost). Internet storage applications require only simple equality predicates, but have strict requirements on correctness and availability (so that data is not missed or lost). Digital library applications require complex search predicates such as range predicates (to search for articles within a date range), but do not have strict correctness and availability requirements. The most demanding applications are transaction processing and military applications, which require both complex range predicates (to search for objects within a region) and strong correctness/availability guarantees.

As an example, consider the Joint Battlespace Infosphere (JBI)[17], a military application that has high scalability and fault-tolerance requirements. One of the potential uses of the JBI is to track information objects, which could include objects in the field such as enemy vehicles. A natural way to achieve the desired scalability and fault-tolerance is to store such objects as (value,item) pairs in a P2P index, where the value could represent the geographic location of the object (in terms of its latitude and longitude), and the item could be a description of that object. Clearly, the JBI requires support for range queries in order to find objects in a certain region. The JBI also requires strong correctness guarantees (so that objects are not missed by a query) and availability guarantees (so that stored objects are not lost).

Current P2P indices, however, do not satisfy the above application needs: while there has been some work on devising P2P indices that can handle expressive range predicates [1, 2, 5, 6, 10, 12, 14, 15, 30], there has been little or no work on guaranteeing correctness and availability in such indices. Specifically, we are not aware of any P2P range index that *guarantees* that a query will not miss items relevant to a query. In fact, we shall later show scenarios whereby range indices [5, 6, 12] that are based on the Chord ring [31] (originally devised for equality queries) can miss query results for range queries, even when the index is operational. Similarly, we are not aware of any range index that can provide provable guarantees on system and item availability.

In this paper, we devise techniques that can provably guarantee query correctness, system availability and item availability in P2P range indices. At a high level, there are two approaches for guaranteeing correctness and availability. The first approach is to simply let the application handle the correctness and availability issues – this, for instance, is the approach taken by CFS [9] and PAST [29], which are applications built on top of the P2P equality indices Chord [31] and Pastry [28], respectively. However, this approach does not work in general for range indices because the application does not (and should not!) have control over various concurrent operations in a P2P range index, including index reorganization and peer failures. Moreover, this approach exposes low-level concurrency details to applications and is also very error-prone due to subtle concurrent interactions between system components.

We thus take the alternative approach of developing new correctness and availability primitives that can be directly implemented in a P2P index. Specifically, we build upon the P2P indexing framework proposed by Crainiceanu et al. [7], and embed novel techniques for ensuring correctness and availability directly into this framework. The benefits of this approach are that it abstracts away the dynamics of the underlying P2P system and provides applications with a consistent interface with provable correctness and availability guarantees. To the best of our knowledge, this is the first attempt to address these issues for both equality and range queries in a P2P index.

One of the benefits of implementing our primitives in the context of a P2P indexing framework is that our techniques are not just applicable to one specific P2P index, but are applicable to all P2P indices that can be instantiated in the framework, including [5, 6, 12]. As a specific instantiation, we implement P-Ring [6], a P2P index that supports both equality and range queries, and show how it can be extended to provide correctness and availability guarantees. We also quantitatively demonstrate the feasibility of our proposed techniques using a real distributed implementation of P-Ring.

The rest of the paper is organized as follows. In Section 2, we present some background material, and in Section 3, we outline our correctness and availability goals. In section 4 we present techniques for guaranteeing query correctness, and in Section 5, we outline techniques for guaranteeing system and item availability. In Section 6, we present our experimental results. In Section 7, we discuss related work, and we conclude in Section 8.

2 Background

In this section, we first introduce our system model and the notion of a *history* of operations, which are used later in the paper. We then briefly review the indexing framework proposed by Crainiceanu et al.[7], and give an example instantiation of this framework for completeness. We use this instantiation in the rest of the paper to discuss problems with existing approaches and to illustrate our newly proposed techniques. We use the framework since it presents a clean way to abstract out different components of a P2P index, and it allows us to confine concurrency and consistency problems to individual components of the framework.

2.1 System Model

A *peer* is a processor with shared storage space and private storage space. The shared space is used to store the distributed data structure for speeding up the evaluation of user queries. We assume that each peer can be identified by a physical id (for example, its IP address). We also assume a fail-stop model for peer failures. A *P2P system* is a collection of peers. We assume there is some underlying network protocol that can be used to send messages reliably from one peer to another with known bounded delay. A peer can join a P2P system by contacting some peer that is already part of the system. A peer can leave the system at any time without contacting any other peer.

We assume that each (data) item stored in a peer exposes a *search key value* from a totally ordered domain $\mathcal{K}$ that is indexed by the system. The search key value for an item i is denoted by $i.skv$. Without loss of generality, we assume that search key values are unique (duplicate values can be made unique by appending the physical id of the peer where the value originates and a version number; this transformation is transparent to users). Peers inserting items into the system can retain ownership of their items. In this case, the items are stored in the private storage partition of the peer, and only pointers to the items are inserted into the system. In the rest of the paper we make no distinction between items and pointers to items.

The queries we consider are range queries of the form $[lb, ub]$, $(lb, ub]$, $[lb, ub)$ or (lb, ub) where $lb, ub \in \mathcal{K}$. Queries can be issued at any peer in the system.

To specify and reason about the correctness and availability guarantees, we use the notion of a *history* of operations [4, 24].

Definition 1 (History $\mathcal{H}$): History $\mathcal{H}$ is a pair $(O, \leq)$ where O is a set of operations and $\leq$ is a partial order defined on these operations.

Conceptually, the partial order $\leq$ defines a *happened before* relationship among operations. If $op_1, op_2 \in O$ are two different operations in history $\mathcal{H}$, and $op_1 \leq op_2$, then intuitively, op_1 finished before op_2 started, i.e., op_1 happened before op_2. If op_1 and op_2 are not related by the partial order, then op_1 and op_2 could have been executed in parallel.

To present our results we also need the notion of a *truncated history* which is a history that only contains operations that happened before a certain operation.

Definition 2 (Truncated History $\mathcal{H}_o$): Given a history $\mathcal{H} = (O_\mathcal{H}, \leq_\mathcal{H})$ and an operation $o \in O_\mathcal{H}$, $\mathcal{H}_o = (O_{\mathcal{H}_o}, \leq_{\mathcal{H}_o})$ is a *truncated history* if $O_{\mathcal{H}_o} = \{o' \in O_\mathcal{H} | o' \leq_\mathcal{H} o\}$ and $\forall o_1, o_2 \in O_{\mathcal{H}_o}$ ($o_1 \leq_\mathcal{H} o_2 \Rightarrow o_1 \leq_{\mathcal{H}_o} o_2$).

2.2 The P2P Indexing Framework From [7]

A P2P index needs to reliably support the following operations: search, item insertion, item deletion, peers join-

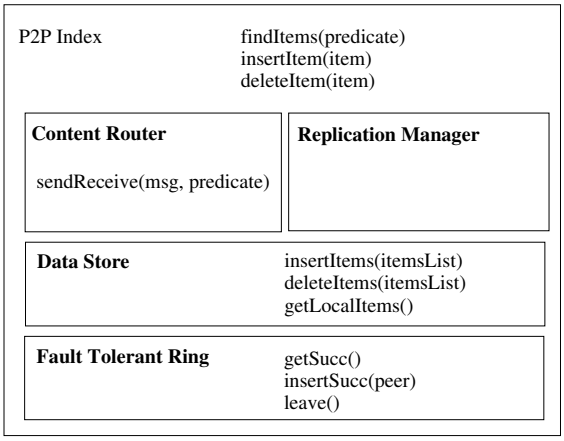

Figure 1. Indexing Framework

ing, and peers leaving the system. We now briefly survey the modularized indexing framework from [7], which is designed to capture most structured P2P indices. Figure 1 shows the components of the framework, and their APIs. The framework does not specify *implementations* for these components but only specifies *functional requirements*.

Fault Tolerant Torus. The Fault Tolerant Torus connects the peers in the system on a torus, and provides reliable connectivity among these peers even in the face of peer failures. For the purposes of this paper, we focus on a Fault Tolerant Ring (a one-dimensional torus). On a ring, for a peer p, we can define the *successor* $\texttt{succ}(p)$ (respectively, *predecessor* $\texttt{pred}(p)$) to be the peer adjacent to p in a clockwise (resp., counter-clockwise) traversal of the ring. Figure 2 shows an example of a Fault Tolerant Ring. If peer p_1 fails, then the ring will reorganize such that $\texttt{succ}(p_5) = p_2$, so the peers remain connected. In addition to maintaining successors, each peer p in the ring is associated with a value, $p.val$, from a totally ordered domain $\mathcal{PV}$. This value determines the position of a peer in the ring, and it increases clockwise around the ring (wrapping around at the highest value). The values of the peers in Figure 2 are shown in parenthesis. The value of a peer is introduced only for ease of exposition and is not required in the formal definition of a ring.

Figure 1 shows the Fault Tolerant Ring API. When invoked on a peer p, $p.\texttt{getSucc}$ returns the address of $\texttt{succ}(p)$. $p.\texttt{insertSucc}(p')$ makes p' the successor of p. $p.\texttt{leave}$ allows p to gracefully leave the ring (of course, p can leave the ring without making this call due to a failure). The ring also exposes events that can be caught at higher layers, such as successor changes (not shown in the figure). An API method need not return right away because of locks and other concurrency issues. Each of the API methods is therefore associated with a start and an end operation. For example, $initLeave(p)$ is the operation associated with the invocation of the API method $p.\texttt{leave()}$ and $leave(p)$ is the operation used to signal the end of this API method. All the operations associated with the initiation and completion of the API methods, as well as the operations associated

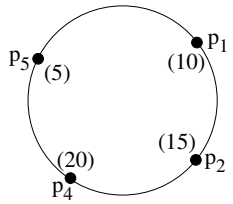

Figure 2: Ring

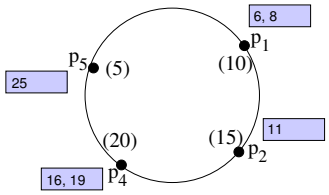

Figure 3: Data Store

with the events raised by the ring form a history called an *API Ring history*. The details can be found in the appendix.

Data Store. The Data Store is responsible for distributing and storing items at peers. The Data Store has a map $\mathcal{M}$ that maps the search key value $i.skv$ of each item i to a value in the domain $\mathcal{PV}$ (the domain of peer values). An item i is stored in a peer p such that $\mathcal{M}(i.skv) \in (\texttt{pred}(p).val, p.val]$. In other words, each peer p is responsible for storing data items mapped to a value between $\texttt{pred}(p).val$ and $p.val$. We refer to the range $(\texttt{pred}(p).val, p.val]$ as $p.range$. We denote the items stored at peer p as $p.items$.

Figure 3 shows an example Data Store that maps some search key values to peers on the ring. For example, peer p_4 is responsible for search key values 16 and 19. One of the main responsibilities of the Data Store is to ensure that the data distribution is uniform so that each peer stores about the same number of items. Different P2P indices have different implementations for the Data Store (e.g., based on hashing [31], splitting, merging and/or redistributing [6, 12]) for achieving this storage balance. As shown in Figure 1, the Data Store provides API methods to insert items into and delete items from the system. It also provides the API method $p.\texttt{getLocalItems()}$ to get the items stored locally in peer p's Data Store.

As with the API Ring History, we can define the *API Data Store history* using the operations associated with the Data Store API methods. Given an API Data Store History $\mathcal{H}$ and a peer p, we use $range_{\mathcal{H}}(p)$ to denote $p.range$ in $\mathcal{H}$ and $items_{\mathcal{H}}(p)$ to denote $p.items$ in $\mathcal{H}$.

Replication Manager. The Replication Manager is responsible for reliably storing items in the system even in the presence of failures, until items are explicitly deleted. As an example, in Figure 5, peer p_1 stores items i_1 and i_2 such that $\mathcal{M}(i_1.skv) = 8$ and $\mathcal{M}(i_2.skv) = 9$. If p_1 fails, these items would be lost even though the ring would reconnect after the failure. The goal of the replication manager is to handle such failures for example by replicating items so that they can be "revived" even if peers fail.

Content Router. The Content Router is responsible for efficiently routing messages to relevant peers in the P2P system. As shown in the API (see Figure 1), the relevant peers are specified by a content-based predicate on search key values, and not by the physical peer ids. This abstracts away the details of storage and index reorganization from higher level applications.

P2P Index. The P2P Index is the index exposed to the end

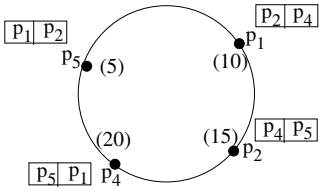

Figure 4: Chord Ring

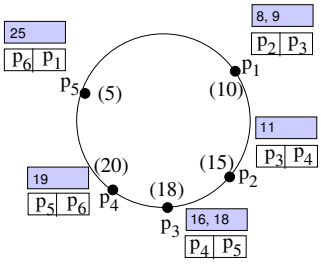

Figure 5: P-Ring Data Store

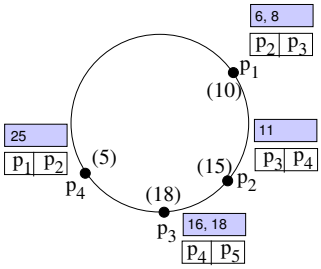

Figure 6: Data Store Merge

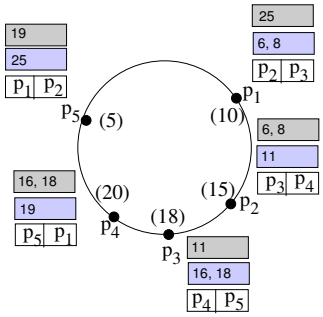

Figure 7: CFS Replication

user. It supports search functionality by using the functionality of the Content Router, and supports item insertion and deletion by using the functionality of the Data Store. As with the API Ring History and API Data Store History, we can define the *API Index History* using the operations associated with the Index API methods.

2.3 An Example Instantiation

We now discuss the instantiation of the above framework using P-Ring [6], an index structure designed for range queries in P2P systems. P-Ring uses the Fault Tolerant Ring of Chord and the Replication Manager of CFS, and only devises a new Data Store and a Content Router for handling data skew. While the full details of P-Ring are presented in [6], we concentrate only on features of P-Ring that are common to many P2P range query index structures from the literature [5, 6, 12]: splitting, merging, and redistributing in order to balance the number of items at each peer. We would like to emphasize that while we use P-Ring as a running example to illustrate query correctness, concurrency, and availability issues in subsequent sections, our discussion also applies to other P2P range indices proposed in the literature.

Fault Tolerant Ring. P-Ring uses the Chord Ring to maintain connectivity among peers [31]. The Chord Ring achieves fault-tolerance by storing a *list* of successors at each peer, instead of storing just a single successor. Thus, even if the successor of a peer p fails, p can use its successor list to identify other peers to re-connect the ring and to maintain connectivity. Figure 4 shows an example Chord Ring in which successor lists are of length 2 (i.e., each peer p stores $\texttt{succ}(p)$ and $\texttt{succ}(\texttt{succ}(p))$ in its successor list). The successor lists are shown in the boxes next to the associated peers. Chord also provides a way to maintain these successor lists in the presence of failures by periodically *stabilizing* a peer p with its first live successor in the successor list. P-Ring also uses Chord to maintain connectivity.

Data Store. Ideally, we would like data items to be uniformly distributed among peers so that the storage load of each peer is about the same. Most existing P2P indices achieve this goal by *hashing* the search key value of an item, and assigning the item to a peer based on this hashed value. Such an assignment is, with high probability, very close to a uniform distribution of entries [27, 28, 31]. However, hashing destroys the value ordering among the search key values, and thus cannot be used to process range queries efficiently (for the same reason that hash indices cannot be used to handle range queries efficiently).

To solve this problem, range indices assign data items to peers directly based on their search key value (i.e., the map $\mathcal{M}$ is order-preserving, in the simplest case it is the identity function). In this case, the ordering of peer values is the same as the ordering of search key values, and range queries can be answered by scanning along the ring. The problem is that now, even in a stable P2P system with no peers joining or leaving, some peers might become overloaded or underloaded due to skewed item insertions and/or deletions. There is a need for a way to dynamically reassign and maintain the ranges associated to the peers. Range indices achieve this goal by `splitting`, `merging` and `redistributing` for handling item overflows and underflows in peers. Let us give an example in the context of P-Ring.

The P-Ring Data Store has two types of peers: *live* peers and *free* peers. Live peers can be part of the ring and store data items, while free peers are maintained separately in the system and do not store any data items.[1] The Data Store ensures that the number of items stored in each live peer is between `sf` and $2 \cdot$ `sf`, where `sf` is some *storage factor*, in order to balance storage between peers.

Whenever the number of items in a peer p's Data Store becomes larger than $2 \cdot$ `sf` (due to many insertions into $p.range$), it is said that an *overflow* occurred. In this case, p tries to *split* its assigned range (and implicitly its items) with a free peer, and to give a fraction of its items to the new peer. Whenever the number of entries in p's Data Store becomes smaller than `sf` (due to deletions from $p.range$), it is said that an *underflow* occurred. In this case, p tries to *merge* with its successor in the ring to obtain more entries. In this case, the successor either *redistributes* its items with p, or gives up its entire range to p and becomes a free peer.

As an illustration of a split, consider the Data Store shown in Figure 3. Assume that `sf` is 1, so each peer can have 1 or 2 entries. Now, when an item i such that $i.skv = 18$ is inserted into the system, it will be stored in

[1] In the actual P-Ring Data Store, free peers also store data items temporarily for some live peers. The ratio of the number of items between any two peers can be bounded, but these details are not relevant in the current context.

p_4, leading to an overflow. Thus, $p_4.range$ will be split with a free peer, and p_4's items will be redistributed accordingly. Figure 5 shows the Data Store after the split, where p_4 split with the free peer p_3, and p_3 takes over part of the items p_4 was originally responsible for (the successor pointers in the Chord Ring are also shown in the figure for completeness). As an illustration of merge, consider again Figure 5 and assume that item i with $t.skv = 19$ is deleted from the system. In this case, there is an underflow at p_4, and p_4 merges with its successor, p_5 and takes over all of p_5's items; p_5 in turn becomes a free peer. Figure 6 shows the resulting system.

Replication Manager. P-Ring uses CFS Replication which works as follows. Consider an item i stored in the Data Store at peer p. The Replication Manager replicates i to k successors of p. In this way, even if p fails, i can be recovered from one of the successors of p. Larger values of k offer better fault-tolerance but have additional overhead. Figure 7 shows a system in which items are replicated with a value of $k = 1$ (the replicated values are shown in the top most box next to the peer).

Content Router. The P-Ring Content Router is based on idea of constructing a hierarchy of rings that can index skewed data distributions. The details of the content router are not relevant here.

3 Goals

We now turn to the main focus of this paper: guaranteeing correctness and availability in P2P range indices. At a high level, our techniques enforce the following design goals.

- **Query Correctness:** A query issued to the index should return all and only those items in the index that satisfy the query predicate.
- **System Availability:** The availability of the index should not be reduced due to index maintenance operations (such as splits, merges, and redistributions).
- **Item Availability:** The availability of items in the index should not be reduced due to index maintenance operations (such as splits, merges, and redistributions).

While the above requirements are simple and natural, it is surprisingly hard to satisfy them in a P2P system. Thus, one approach is to simply leave these issues to higher level applications – this is the approach taken by CFS [9] and PAST [29], which are applications built on top of Chord [31] and Pastry [28], respectively, two index structures designed for equality queries. The downside of this approach is that it becomes quite complicated for application developers because they have to understand the details of how lower layers are implemented, such as how ring stabilization is done. Further, this approach is also error-prone because complex concurrent interactions between the different layers (which we illustrate in Section 4) make it difficult to devise a system that produces consistent query results. Finally, even if application developers are willing to take responsibility for the above properties, there are no known techniques for ensuring the above requirements for P2P range indices.

In contrast, the approach we take is to cleanly encapsulate the concurrency and consistency aspects in the different layers of the system. Specifically, we embed consistency primitives in the Fault Tolerant Ring and the Data Store, and provide handles to these primitives for the higher layers. With this encapsulation, higher layers and applications can simply use these APIs without having to explicitly deal with low-level concurrency issues or knowing how lower layers are implemented, while still being guaranteed query consistency and availability for range queries.

Our proposed techniques differ from distributed database techniques [20] in terms of scale (hundreds to thousands of peers, as opposed to a few distributed database sites), failures (peers can fail at any time, which implies that blocking concurrency protocols cannot be used), and perhaps most importantly, dynamics (due to unpredictable peer insertions and deletions, the location of the items is not known a priori and can change *during* query processing).

In the subsequent two sections, we describe our solutions to query correctness and system and item availability.

4 Query Correctness

We focus on query consistency for range queries (note that equality queries are a special case of range queries). We first formally define what we mean by query correctness in the context of the indexing framework. We then illustrate scenarios where query correctness can be violated if we directly use existing techniques. Finally, we present our solutions to these problems. Detailed definitions and proofs for all theorems stated in this section can be found in the appendix.

4.1 Defining Correct Query Results

Intuitively, a system returns a correct result for a query Q if and only if the result contains all and only those items in the system that satisfy the query predicate. Translating this intuition into a formal statement in a P2P system requires us to define which items are "in the system"; this is more complex than in a centralized system because peers can fail, can join, and items can move between peers during the duration of a query. We start by defining an index P as a set of peers $P = \{p_1, \ldots, p_n\}$, where each peer is structured according to the framework described in Section 2.2. To capture what it means for an item to be in the system, we now introduce the notion of a *live item*.

Definition 3 (Live Item): An item i is *live* in API Data Store History $\mathcal{H}$, denoted by $live_{\mathcal{H}}(i)$, iff $\exists p \in P\ (i \in items_{\mathcal{H}}(p))$.

In other words, an item i is live in API Data Store History $\mathcal{H}$ iff the peer with the appropriate range contains i in its Data Store. Given the notion of a live item, we can define a correct query result as follows. We use $satisfies_Q(i)$ to denote whether item i satisfies query Q's query predicate.

Definition 4 (Correct Query Result): Given an API Data Store History $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, a set R of items is a *correct query result* for a query Q initiated with operation o_s and

successfully completed with operation o_e iff the following two conditions hold:

1. $\forall i \in R$ ($satisfies_Q(i) \wedge \exists o \in O_{\mathcal{H}}$ ($o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i)$))
2. $\forall i$ ($satisfies_Q(i) \wedge \forall o \in O_{\mathcal{H}}(o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i) \Rightarrow i \in R$)).

The first condition states that only items that satisfy the query predicate and which were live at some time during the query evaluation should be in the query result. The second condition states that all items that satisfy the query predicate and which were live throughout the query execution must be in the query result.

4.2 Incorrect Query Results: Scenarios

Existing index structures for range queries evaluate a range query in two steps: (a) finding the peer responsible for left end of the query range, and (b) scanning along the ring to retrieve the items in the range. The first step is achieved using an appropriate Content Router, such as SkipGraphs [2] or the P-Ring [6] Content Router, and the related concurrency issues have been described and solved elsewhere in the literature [2, 6]. We thus focus on the second step (scanning along the ring) and show how existing techniques can produce incorrect results.

Scanning along the ring can produce incorrect query results due to two reasons. First, the ring itself can be temporarily inconsistent, thereby skipping over some live items. Second, even if the ring is consistent, concurrency issues in the Data Store can sometimes result in incorrect results. We now illustrate both of these cases using examples.

4.2.1 Inconsistent Ring Consider the Ring and Data Store shown in Figure 5. Assume that item i with $\mathcal{M}(i.skv) = 6$ is inserted into the system. Since $p_1.range = (5, 10]$, i will be stored in p_1's Data Store. Now assume that p_1's Data Store overflows due to this insertion, and hence p_1 splits with a new peer p and transfers some of its items to p. The new state of the Ring and Data Store is shown in Figure 8. At this point, $p.range = (5, 6]$ and $p_1.range = (6, 10]$. Also, while p_5's successor list is updated to reflect the presence of p, the successor list of p_4 is not yet updated because the Chord ring stabilization proceeds in rounds, and p_4 will only find out about p when it next stabilizes with its successor (p_5) in the ring.

Now assume that p_5 fails. Due to the Replication Manager, p takes over the range $(20, 6]$ and adds the data item i such that $\mathcal{M}(i.skv) = 25$ into its Data Store. The state of the system at this time is now shown in Figure 9. Now assume that a search Q originates at p_4 for the range $(20, 9]$. Since $p_4.val$ is the lower bound of the query range, p_4 tries to forward the message to the first peer in its successor list (p_5), and on detecting that it has failed, forwards it to the next peer in its successor list (p_1). p_1 returns the items in the range $(6, 10]$, but the items in the range $(20, 6]$ are missed! (Even though all items in this range are live – they are in p's Data Store.) This problem arises because the successor pointers for p_4 are temporarily inconsistent during the insertion of p (they point to p_1 instead of p). Eventually, of course, the ring will stabilize and p_4 will point to p as its successor, but *before* this ring stabilization, query results can be missed.

At this point, the reader might be wondering whether a simple "fix" might address the above problem. Specifically, what if p_1 simply rejects the search request from p_4 (since p_4 is not p_1's predecessor) until the ring stabilizes? The problem with this approach is that p_1 does not know whether p has also failed, in which case p_4 is indeed p_1's predecessor, and it should accept the message. Again, the basic problem is that a peer does not have precise information about other peers in the system (due to the dynamics of the P2P system), and hence potential inconsistencies can occur. We note that the scenario outlined in Figure 9 is just one example of inconsistencies that can occur in the ring – rings with longer successor lists can have other, more subtle, inconsistencies (for instance, when p is not the direct predecessor of p_1).

4.2.2 Concurrency in the Data Store We now show how concurrency issues in the Data Store can produce incorrect query results, *even if the ring is fully consistent*. We illustrate the problem in the context of a Data Store redistribute operation; similar problems arise for Data Store splits and merges.

Consider again the system in Figure 5 and assume that a query Q with query range $(10, 18]$ is issued at p_2. Since the lower bound of $p_2.range$ is the same as the lower bound of the query range, the sequential scan for the query range starts at p_2. The sequential scan operation first gets the data items in p_2's Data Store, and then gets the successor of p_2 in the ring, which is p_3. Now assume that the item i with $\mathcal{M}(i.skv) = 11$ is deleted from the index. This causes p_2 to become underfull (since it has no items left in its Data Store), and it hence redistributes with its successor p_3. After the redistribution, p_2 becomes responsible for the item i_1 with $\mathcal{M}(i_1.skv) = 16$, and p_3 is no longer responsible for this item. The current state of the index is shown in Figure 10.

Now assume that the sequential scan of the query resumes, and the scan operation propagates the scan to p_3 (the successor of p_2). However, the scan operation will miss item i_1 with $\mathcal{M}(i_1.skv) = 16$, even though i_1 satisfies the query range and was live throughout the execution of the query! This problem arises because of the concurrency issues in the Data Store – the range that p_2's Data Store was responsible for changed while p_2 was processing a query. Consequently, some query results were missed.

4.3 Ensuring Correct Query Results

We now present solutions that avoid the above scenarios and provably guarantee that the sequential scan along the ring for range queries will produce correct query results. The attractive feature of our solution is that these enhancements are confined to the Ring and Data Store components of the architecture, and higher layers (both applications on top of the P2P system and other components of the P2P sys-

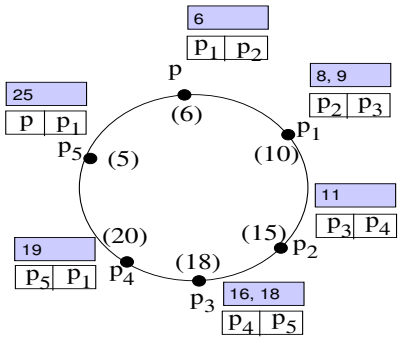

Figure 8. Peer p just inserted into the system

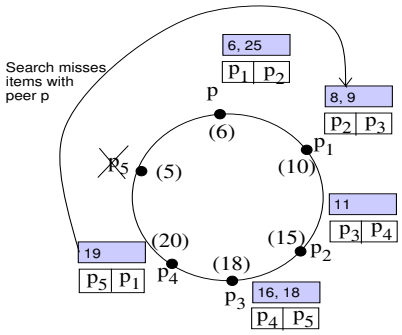

Figure 9. Incorrect query results: Search Q originating at peer p_4 misses items in p

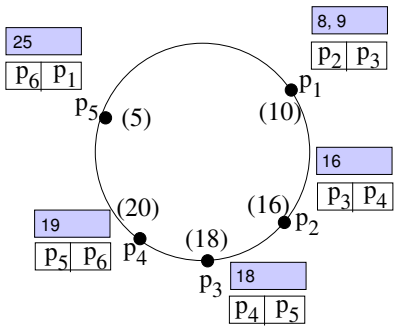

Figure 10. System after peer p_2 redistributes with peer p_3

tem itself) can be guaranteed correctness by accessing the components through the appropriate API. We first present a solution that addresses ring inconsistency, and then present a solution that addresses Data Store concurrency issues.

4.3.1 Handling Ring Inconsistency As illustrated in Section 4.2.1, query results can be incorrect if a peer's successor list pointers are temporarily inconsistent (we shall formally define the notion of consistency soon). Perhaps the simplest way to solve this problem is to explicitly avoid this inconsistency by atomically updating the successor pointers of every relevant peer during each peer insertion. For instance, in the example in Section 4.2.1, we could have avoided the inconsistency if p_5's *and* p_4's successor pointers had been atomically updated during p's insertion. Unfortunately, this is not a viable solution in a P2P system because there is no easy way to determine the peers whose successor lists will be affected by an insertion since other peers can concurrently enter, leave or fail, and any cached information can become outdated.

To address this problem, we introduce a new method for implementing `insertSucc` (Figure 1) that ensures that successor pointers are always consistent even in the face of concurrent peer insertions and failures (peer deletions are considered in the next section). Our technique works asynchronously and does not require any up-to-date cached information or global co-ordination among peers. The main idea is as follows. Each peer in the ring can be in one of two states: `JOINING` or `JOINED`. When a peer is initially inserted into the system, it is in the `JOINING` state. Pointers to peers in the `JOINING` state need not be consistent. However, each `JOINING` peer transitions to the `JOINED` state in some bounded time. We ensure that the successor pointers to/from `JOINED` peers are always consistent. The intuition behind our solution is that a peer p remains in the `JOINING` state until all relevant peers know about p – it then transitions to the `JOINED` state. Higher layers, such as the Data Store, only store items in peers in the `JOINED` state, and hence avoid inconsistencies.

We now formally define the notion of consistent successor pointers. We then present our distributed, asynchronous algorithm for `insertSucc` that satisfies this property for `JOINED` peers.

4.3.1.1 Defining Consistent Successor Pointers

We first introduce some notation. Let $\mathcal{H}$ be a given API Ring History. This history induces a ring, denoted by $R_{\mathcal{H}}$. Let $P_{\mathcal{H}}$ be the set of live peers in `JOINED` state in the ring. $p.succList_{\mathcal{H}}$ is the successor list of peer p in $\mathcal{H}$. $p.succList_{\mathcal{H}}.length$ is the length (number of pointers) of $p.succList_{\mathcal{H}}$, and $p.succList_{\mathcal{H}}[i]$ ($0 \leq i < p.succList_{\mathcal{H}}.length$) refers to the i'th pointer in $succList$. We define $p.trimList_{\mathcal{H}}$ as the *trimmed* copy of $p.succList_{\mathcal{H}}$ with only pointers corresponding to live peers in `JOINED` state in $R_{\mathcal{H}}$.

Definition 5 (Consistent Successor Pointers): Given an API Ring History $\mathcal{H}$, the ring $R_{\mathcal{H}}$ induced by $\mathcal{H}$ has *consistent successor pointers* iff the following condition holds:

- $\forall p \in \mathcal{P}_{\mathcal{H}}$ ($\forall i$ ($0 \leq i < p.trimList_{\mathcal{H}}.length \Rightarrow succ_{\mathcal{H}}(p.trimList_{\mathcal{H}}[i]) = p.trimList_{\mathcal{H}}[i+1]$) $\wedge$ $succ_{\mathcal{H}}(p) = p.trimList_{\mathcal{H}}[0]$).

The above definition says that there are no peers in the ring between consecutive entries of $p.trimList$ i.e. p cannot have "missing" pointers to peers in the set $\mathcal{P}_{\mathcal{H}}$. In our example in Figure 8, the successor pointers are not consistent with respect to the set of all peers in the system because p_4 has a pointer to p_5 but not to p.

4.3.1.2 Proposed Algorithm

We first present the intuition behind our insert algorithm. Assume that a peer p' is to be inserted as the successor of a peer p. Initially, p' will be in the `JOINING` state. Eventually, we want p' to transition to the `JOINED` state, without violating the consistency of successor pointers. According to the definition of consistent successor pointers, the only way in which converting p' from the `JOINING` state to the `JOINED` state can violate consistency is if there exist `JOINED` peers p_x and p_y such that: $p_x.succList[i] = p$ and $p_x.succList[i+k] = p_y$ (for some $k > 1$) and for all $j, 0 < j < k$, $p_x.succList[i+j] \neq p'$. In other words, p_x has pointers to p and p_y but not to p' whose value occurs between $p.val$ and $p_y.val$.

Algorithm 1 : p_1.insertSucc(*Peer* p)

```
0:  // Insert p into lists as a JOINING peer
1:  writeLock succList, stateList
2:  succList.push_front(p)
3:  stateList.push_front(JOINING)
4:  releaseLock stateList, succList
5:  // Wait for successful insert ack
6:  wait for JOIN ack; on ack do:
7:  // Notify p of successful insertion and update lists
8:  writeLock succList, stateList
9:  Send a message to p indicating it is now JOINED
10: stateList.update_front(JOINED)
11: succList.pop_back(), stateList.pop_back()
12: releaseLock stateList, succList
```

Algorithm 2 : Ring Stabilization

```
0:  // Update lists based on successor's lists
1:  readLock succList, stateList
2:  get succList/stateList from first non-failed p_s in
    succList
3:  upgradeWriteLock succList, stateList
4:  succList = p_s.succList; stateList = p_s.stateList
5:  succList.push_front(p_s)
6:  stateList.push_front(JOINED)
7:  succList.pop_back(), stateList.pop_back()
8:  // Handle JOINING peers
9:  listLen = succList.length
10: if stateList[listLen − 1] == JOINING then
11:     succList.pop_back(); stateList.pop_back()
12: else if stateList[listLen − 2] == JOINING then
13:     Send an ack to succList[listLen − 3]
14: end if
15: releaseLock stateList, succList
```

Our algorithm avoids this case by ensuring that at the time p' changes from the JOINING state to the JOINED state, if p_x has pointers to p and p_y (where p_y's pointer occurs after p's pointer), then it also has a pointer to p'. It ensures this property by propagating the pointer to p' to all of p's predecessors until it reaches the predecessor whose last pointer in the successor list is p (which thus does not have a p_y that can violate the condition). At this point, it transitions p' from the JOINING to the JOINED state. Propagation of p' pointer is piggybacked on the Chord ring stabilization protocol, and hence does not introduce new messages.

Algorithms 1 and 2 show the pseudocode for the insertSucc method and the modified ring stabilization protocol, respectively. In the algorithms, we assume that in addition to $succList$, each peer has a list called $stateList$ which stores the state (JOINING or JOINED) of the corresponding peer in $succList$. We walk through the algorithms using an example.

Consider again the example in Figure 5, where p is to be added as a successor of p_5. The insertSucc method is invoked on p_5 with a pointer to p as the parameter. The method first acquires a write lock on

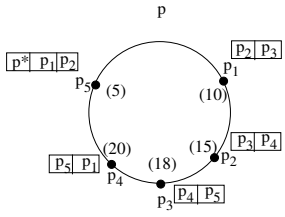

Figure 11: After p_5.insertSucc call

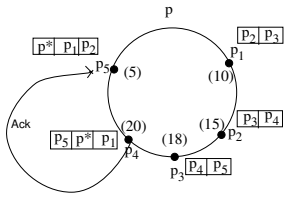

Figure 12: Propagation and final ack

$succList$ and $stateList$, inserts p as the first pointer in $p_5.succList$ (thereby increasing its length by one), and inserts a corresponding new entry into $p_5.stateList$ with value JOINING (lines $2-4$ in Algorithm 1). The method then releases the locks on $succList$ and $stateList$ (line 5) and blocks waiting for an acknowledgment from some predecessor peer indicating that it is safe to transition p from the JOINING state to the JOINED state (line 7). The current state of the system is shown in Figure 11 (JOINING list entries are marked with a "*").

Now assume that a ring stabilization occurs at p_4. p_4 will first acquire a read lock on its $succList$ and $stateList$, contact the first non-failed entry in its successor list, p_5, to get p_5's $succList$ and $stateList$ (lines $2-3$ in Algorithm 2). p_4 then acquires a write lock on its $succList$ and $stateList$, and copies over the $succList$ and $stateList$ it obtained from p_5 (lines $4-5$). p_4 then inserts p_5 as the first entry in $succList$ (increasing its length by 1) and also inserts the corresponding state in $stateList$ (the state will always be JOINED because JOINING nodes do not respond to ring stabilization requests). p_4 then removes the last entries in $succList$ and $stateList$ (lines $6-8$) to ensure that its lists are of the same length as p_5's lists. The current state of the system is shown in Figure 12.

p_4 then checks whether the state of the last entry is JOINING; in this case it simply deletes the entry (lines $11-12$) because it is far enough from the JOINING node that it does not need to know about it (although this case does not arise in our current scenario for p_4). p_4 then checks if the state of the penultimate peer (p) is JOINING – since this is the case in our scenario, p_4 sends a acknowledgment to the peer preceding the penultimate peer in the successor list (p_5) indicating that p can be transitioned from JOINING to JOINED since all relevant predecessors know about p (lines $13-14$). p_4 then releases the locks on its lists (line 16).

The insertSucc method of p_5, on receiving a message from p_4, first send a message to p indicating that it is now in the JOINED state (line 10). p_5 then changes the state of its first list entry (p) to JOINED and removes the last entries from its lists in order to shorten them to the regular length (lines $11-12$). The final state after p is inserted into the ring and multiple ring stabilizations have occurred is shown in Figure 13.

One optimization we implement for the above method is to *proactively* contact the predecessor in the ring whenever

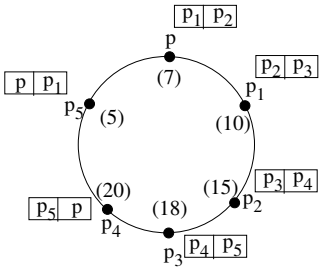

Figure 13: Completed `insertSsucc`

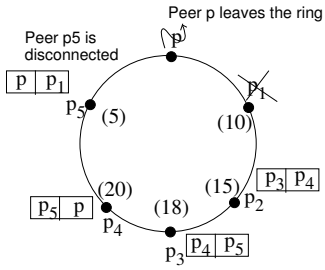

Figure 14: Naive merge leads to reduced reliability

`insertSucc` is in progress, to trigger ring stabilization. This expedites the operation since it is no longer limited by the frequency of the ring stabilization process.

We can define a *PEPPER Ring History* to capture our implementation of the ring API, including the operations in Algorithms 1 and 2. We can prove the following theorem.

Theorem 1 (Consistent Successor Pointers): *Given a PEPPER Ring History $\mathcal{PH}$, the ring $R_{\mathcal{PH}}$ induced by $\mathcal{PH}$ has consistent successor pointers*.

4.3.2 Handling Data Store Concurrency Recall from the discussion in Section 4.2.2 that even if the ring is fully consistent, query results can be missed due to concurrency issues at the Data Store. Essentially, the problem is that the range of a peer can change while a query is in progress, causing the query to miss some results. How do we shield the higher layers from the concurrency details of the Data Store while still ensuring correct query results?

Our solution to this problem is as follows. We introduce a new API method for the Data Store called `scanRange`. This method has the following signature: `scanRange(lb, ub, handlerId, param)`, where (1) *lb* is the lower bound of the range to be scanned, (2) *ub* is the upper bound of the range to be scanned, (3) *handlerId* is the id of the handler to be invoked on every peer p such that $p.range$ intersects $[lb, ub]$ (i.e., p's range intersects the scan range), and (4) *param* is the parameter to be passed to the handlers. The `scanRange` method should be invoked on the Data Store of the peer p_1 such that $lb \in p_1.range$ (i.e., the first peer whose range intersects the scan range). The start and end operations associated with `scanRange` are $initScanRange_i(p_1, lb, ub)$ and $doneScanRange_i(p_n, lb, ub)$ for some $i \in \mathcal{N}$. The index i is used to distinguish multiple invocations of the API method with the same signature. The `scanRange` method causes the appropriate handler to be invoked on every peer p such that $p.range$ intersects $[lb, ub]$. $scanRange_i(p, p_1, r)$ is the operation in the API Data Store History that is associated with the invocation of the appropriate handler at peer p. Here, r is the subset of $p.range$ that intersects with $[lb, ub]$.

`scanRange` handles all the concurrency issues associated with the Data Store. Consequently, higher layers do not have to worry about changes to the Data Store while a scan is in progress. Further, since `scanRange` allows applications to register their own handlers, higher layers can customize the scan to their needs (we shall soon show how we can collect range query results by registering appropriate handlers).

We now introduce some notation before we define the notion of *scanRange correctness*. We use $scanOps(i)$ to denote the set of $scanRange_i(p, p_1, r)$ operations associated with the i^{th} invocation of `scanRange`. We use $rangeSet(i) = \{r | \exists p_1, p_2\ scanRange_i(p_1, p_2, r) \in scanOps(i)\}$ to denote the set of ranges reached by `scanRange`. We use $r_1 \bowtie r_2$ to denote that range r_1 overlaps with range r_2 and we use $r_1 \cup r_2$ to denote the union of range r_1 with range r_2.

We can define *scanRange correctness* as follows:

Definition 6 (scanRange Correctness): An API Data Store History $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ is said to satisfy *scanRange correctness* iff $\forall i \in \mathcal{N}\ \forall lb, ub\ \forall p_1 \in \mathcal{P}\ o_e = doneScanRange_i(p_1, lb, ub) \in O_{\mathcal{H}} \Rightarrow$

1. $o_s = initScanRange_i(p_1, lb, ub) \leq_{\mathcal{H}} o_e$
2. $\forall o \in scanOps(i)\ \forall p\ \forall r\ o = scanRange_i(p, p_1, r) \Rightarrow o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge r \subseteq range_{\mathcal{H}_o}(p)$
3. $\forall o_l, o_m \in scanOps(i)\ o_l \neq o_m \wedge \forall p_l, p_m\ \forall r_l, r_m\ o_l = scanRange_i(p_l, p_1, r_l) \wedge o_m = scanRange_i(p_m, p_1, r_m) \Rightarrow \neg(o_l \bowtie o_m)$
4. $[lb, ub] = \cup_{r \in rangeSet(i)}(r)$

Condition 1 states that the initiate operation for `scanRange` should occur before the completion operation. Condition 2 states that range r used to invoke the handler at peer p is a subset of p's range. Condition 3 states that ranges r_l and r_m used to invoke the handlers at distinct peers p_l and p_m, respectively, are non-overlapping. Finally, condition 4 states that the union of all ranges used to invoke the handlers is $[lb, ub]$.

4.3.2.1 Implementing scanRange

We present now our implementation for the `scanRange` API method. Algorithm 3 shows the pseudocode for the `scanRange` method executed at a peer p. The method first acquires a read lock on the Data Store $range$ (to prevent it from changing) and then checks to make sure that $lb \in p.range$, i.e., p is the first peer in the range to be scanned (lines 1-2). If the check fails, `scanRange` is aborted (lines 3-4). If the check succeeds, then the helper method `processHandler` is invoked. `processHandler` (Algorithm 4) first invokes the appropriate handler for the scan (lines 1-3), and then checks to see whether the scan has to be propagated to p's successor (line 4). If so, it invokes the `processScan` method on p's successor.

Algorithm 5 shows the code that executes when p_{succ}.`processScan` is invoked by p.`processHandler`. `processScan` *asynchronously* invokes the `processHandler` method on p_{succ}, and returns. Consequently, p holds on to a lock on its range only until p_{succ} locks its range; once p_{succ} locks its range, p can release its lock, thereby allowing for

Algorithm 3 : $p.scanRange(lb, ub, handlerId, param)$

```
0: readLock range
1: if lb ∉ p.range then
2:    // Abort scanRange
3:    releaseLock range
4: else
5:    // p is the first peer in scan range
6:    p.processHandler(r, handlerId, param)
7: end if
```

Algorithm 4 : $p.processHandler(lb, ub, handlerId, param)$

```
0: // Invoke appropriate handler with relevant range r
1: Get handler with id handlerId
2: r = [lb, ub] ∩ p.range
3: newParam = handler.handle(r, param)
4: // Forward to successor if required
5: if ub ∉ p.range then
6:    p_succ = p.ring.getSucc()
7:    p_succ.processScan(lb, ub, handlerId, newParam)
8: end if
9: releaseLock range
```

Algorithm 5 : $p.processScan(lb, ub, handlerId, param)$

```
0: readLock range
1: Invoke p.processHandler(lb, ub, handlerId, param)
   asynchronously
2: return
```

more concurrency. Note that p can later split, merge, or redistribute, but this will not produce incorrect query results since the scan has already finished scanning the items in p.

We now illustrate the working of these algorithms using an example. Assume that `scanRange`$(10, 18, h_1, param_1)$ is invoked in p_2 in Figure 5. p_2 locks its range in `scanRange` (to prevent p_2's range from changing), invokes the handler corresponding to h_1 in `processHandler`, and then invokes `processScan` on p_3. p_3 locks its range in `processScan`, asynchronously invokes `processHandler` and returns. Since p_3.`processScan` returns, p_2 can now release its lock and participate in splits, merges, or redistributions. However, p_3 holds onto a lock on its range until p_3 handler is finished executing. Thus, the algorithms ensure that a peer's range does not change during a scan, but releases locks as soon as the scan is propagated to the peer's successor, for maximum concurrency.

We can define a *PEPPER Data Store History* to capture our implementation of the Data Store API augmented with the new operation `scanRange`. We can prove the following correctness theorem.

Theorem 2 (scanRange Correctness): *Any PEPPER Data Store History satisfies the scanRange correctness property.*

Using the `scanRange` method, we can easily ensure correct results for range queries by registering the appropriate handler. Algorithm 6 shows the algorithm for evaluating range queries. lb and ub represent the lower and upper bounds of the range to be scanned, and pid represents the id of the peer to which the final result is to be sent. As shown, the algorithm simply invokes the `scanRange` method with parameters lb, ub, the id of the range query handler, and a parameter for that handler. The id of the peer pid that the result should be sent to is passed as a parameter to the range query handler. The range query handler (Algorithm 7) invoked with range r at a peer p works as follows. It first gets the items in p's Data Store that are in range r and hence satisfy the query result (lines 1-2). Then, it sends the items and the range r to the peer pid (line 3).

Algorithm 6 : $p.rangeQuery(lb, ub, pid)$

```
0: // Initiate a scanRange
1: p.scanRange(lb, ub, rangeQueryHandlerId, pid)
```

Algorithm 7 : $p.rangeQueryHandler(r, pid)$

```
0: // Get results from p's Data Store
1: Find items in p's Data Store in range r
2: Send < items, r > to peer pid
```

Using the above implementation of a range query, the inconsistency described in Section 4.2.2 cannot occur because p_2's range cannot change (and hence redistribution cannot happen) when the search is still active in p_2. We can prove the following correctness theorem:

Theorem 3 (Search Correctness): *Given a PEPPER Data Store History $\mathcal{PH}$, all query results produced in $\mathcal{PH}$ are* correct *(as per the definition of correct query results in Section 4.1).*

5 System and Item Availability

We now address system availability and item availability issues. Intuitively, ensuring system availability means that the availability of the index should not be reduced due to routine index maintenance operations, such as splits, merges, and redistributions. Similarly, ensuring item availability means that the availability of items should not be reduced due to maintenance operations. Our discussion of these two issues is necessarily brief due to space constraints, and we only illustrate the main aspects and sketch our solutions.

5.1 System Availability

An index is said to be *available* if its Fault Tolerant Ring is connected. The rationale for this definition is that an index can be operational (by scanning along the ring) so long as its peers are connected. The Chord Fault Tolerant Ring provides strong availability guarantees when the only operations on the ring are peer insertions (splits) and failures [31]. These availability guarantees also carry over to our variant of the Fault Tolerant Ring with the new implementation of `insertSucc` described earlier because it is a stronger version of the Chord's corresponding primitive (it satisfies all the properties required for the Chord proofs). Thus, the only index maintenance operation that can reduce

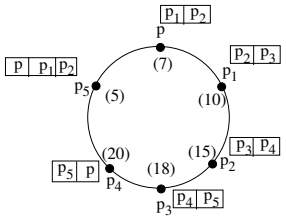

Figure 15: Controlled leave of peer p

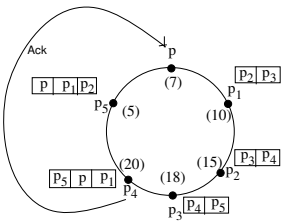

Figure 16: Final ack received at peer p. Peer p is good to go.

the availability of the system is the merge operation in the Data Store, which translates to the `leave` operation in the Fault Tolerant Ring. Note that the redistribute operation in the Data Store does not affect the ring connectivity.

We show that a naive implementation of `leave`, which is simply removing the merged peer from the ring, reduces system availability. We then sketch an alternative implementation for the `leave` that provably does not reduce system reliability. Using this new implementation, the Data Store can perform a merge operation without knowing the details of the ring stabilization, while being guaranteed that system availability is not compromised.

Naive `leave` Reduces System Availability: Consider the system in Figure 13 in which the length of the successor list of each peer is 2. Without a `leave` primitive, this system can tolerate one failure per peer stabilization round without disconnecting the ring (since at most one of a peer's two successor pointers can become invalid before the stabilization round). We now show that in the presence of the naive `leave`, a single failure can disconnect the ring. Thus, `leave` reduces the availability of the system. Assume that `leave` is invoked on p, and p immediately leaves the ring. Now assume that p_1 fails (this is the single failure). The current state of the system is shown in Figure 14, and as we can see, the ring is disconnected since none of p_5's successor pointers point to peers in the ring.

Solution Sketch: The reason the naive implementation of `leave` reduced availability is that pointers to the peer p leaving the ring become invalid. Hence, the successor lists of the peers pointing to p effectively decreases by one, thereby reducing availability. To avoid this problem, our solution is to increase the successor list lengths of all peers pointing to p by one. In this way, when p leaves, the availability of the system is not compromised. As in the `insertSucc` case, we piggyback the lengthening of the successor lists on the ring stabilization protocol. This is illustrated in the following example.

Consider Figure 13 in which `leave` is invoked on p. During the next ring stabilization, the predecessor of p, which is p_5, increases its successor list length by 1. The state of the system is shown in Figure 15. During the next ring stabilization, the predecessor of p_5, which is p_4, increases its successor list length by 1. Since p_4 is the last predecessor that knows about p, p_4 sends a message to p indicating that it is safe to leave the ring. The state of the system at this point is shown in Figure 16. It is easy to see

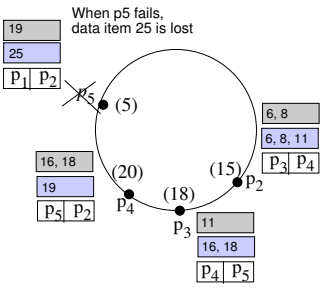

Figure 17: Peer p_5 fails causing loss of item 25

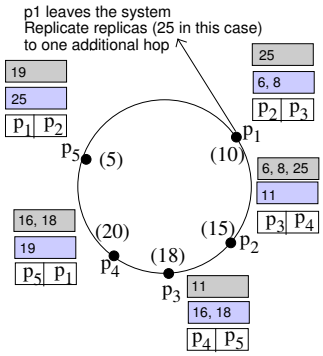

Figure 18: Replicate item 25 one additional hop.

that if p leaves the ring at this point, a single failure cannot disconnect the ring, as in was the case in the previous example. We can formally prove that the new algorithm for `leave` does not reduce the availability of the system.

5.2 Item Availability

We first formalize the notion of item availability in a P2P index.

We represent the successful insertion of an item i at peer p with operation $insertItem(i, p)$ and deletion of an item i' at peer p' with operation $deleteItem(i', p')$.

Definition 7 (Item Availability): Given an API Index History $\mathcal{H}$, an index P is said to preserve *item availability* iff $\forall i$ ($\exists p \in P$ ($insertItem(i, p) \in O_{\mathcal{H}}$) $\wedge$ $\nexists p' \in P$ ($deleteItem(i, p') \in O_{\mathcal{H}}$) $\Rightarrow live_{\mathcal{H}}(i)$).

In other words, if item i has been inserted but not deleted wrt to *API Index history* $\mathcal{H}$ then i is a live item.

The CFS Replication Manager, implemented on top of the Chord Ring provides strong guarantees [9] on item availability when the only operations on the ring are peer insertions and failures, and these carry over to our system too. Thus, the only operation that could compromise item availability is the `leave` operation invoked on a merge. We now show that using the original CFS Replication Manager in the presence of merges does compromise item availability. We then describe a modification to the CFS Replication Manager and its interaction with the Data Store that ensures the original guarantees on item availability.

Scenario that Reduces Item Availability: Consider the system in Figure 7. The top box associated with each peer represents the items replicated at that peer (CFS replicates items along the ring). In this example, each item is replicated to one successor along the ring; hence, the system can tolerate one failure between replica refreshes. We now show how, in the presence of Data Store merges, a single failure can compromise item availability. Assume that peer p_1 wishes to merge with p_2 in Figure 7. p_1 thus performs an `leave` operation, and once it is successful, it transfers its Data Store items to p_2 and leaves the system. The state of the system at this time is shown in Figure 17. If p_5 fails at this time (this is the single failure), the item i such that $\mathcal{M}(i.skv) = 25$ is lost.

Solution Sketch: The reason item availability was compromised in the above example is because when p_1 left the system, the replicas it stored were lost, thereby reducing the number of replicas for certain items in the system. Our solution is to replicate the items stored in the merging peer p's Replication Manager for one additional hop before p leaves the system. This is illustrated in Figure 18, where before p_1 merges with p_2, it creates one more replica for items in its Data Store and Replication Manager, at one additional peer. When p_1 finally merges with p_2 and leaves the system, the number of replicas is not reduced, thereby preserving item availability. We can prove that the above scheme preserves item availability even in the presence of concurrent splits, merges, and redistributions.

6 Experimental Evaluation

We had two main goals in our experimental evaluation: (1) to demonstrate the feasibility of our proposed query correctness and availability algorithms in a dynamic P2P system, and (2) to measure the overhead of our proposed techniques. Towards this goal, we implemented the P-Ring index, along with our proposed correctness and availability algorithms, in a real distributed environment with concurrently running peers. We used this implementation to measure the overhead of each of our proposed techniques as compared to the naive approach, which does not guarantee correctness or availability.

6.1 Experimental Setup

We implemented the P-Ring index as an instantiation of the indexing framework (Section 2.3). The code was written in C++ and all experiments were run on a cluster of workstations, each of which had 1GHz processor, 1GB of main memory and at least 15GB of disk space. All experiments were performed with 30 peers running concurrently on 10 machines (with 3 peers per machine). The machines were connected by a local area network.

We used the following default parameter values for our experiments. The length of the Chord Fault-Tolerant Ring successor list was 4 (which means that the ring can tolerate up to 3 failures without being disconnected if the ring is fully consistent). The ring stabilization period was 4 seconds. We set the storage factor of the P-Ring Data Store to be 5, which means that it can hold between 5 and 10 data items. The replication factor in the Replication Manager is 6, which means that each item is replicated 6 times. We vary these parameters too in some of the experiments.

We ran experiments in two modes of the system. The first mode was the *fail-free* mode, where there were no peers failures (although peers are still dynamically added and splits, merges, and redistributes occur in this state). The second was the *failure* mode, where we introduced peer failures by killing peers. For both modes, we added peers at a rate of one peer every 3 seconds, and data items were added at the rate of 2 items per second. We also vary the rate of peer failures in the failure mode.

6.2 Implemented Approaches

We implemented and evaluated all four techniques proposed in this paper. Specifically, we evaluate (1) the *insertSucc* operation that guarantees ring consistency, (2) the *scanRange* operation that guarantees correct query results, (3) the *leave* operation that guarantees system availability, and (4) the *replication to additional hop* operation that guarantees item availability. For *scanRange*, we implemented a synchronous version where the *processHandler* is invoked synchronously at each peer (see Algorithm 5).

One of our goals was to show that the proposed techniques actually work in a real distributed dynamic P2P system. The other goal was to compare each solution with a naive approach (that does not provide correctness or availability guarantees). Specifically, for the *insertSucc* operation, we compare it with the naive *insertSucc*, where the joining peer simply contacts its successor and becomes part of the ring. For the *scanRange* operation, we compare it with the naive range query method whereby the application explicitly scans the ring without using the *scanRange* primitive. For the *leave* operation, we compare with the naive approach where the peer simply leaves the system without notifying other peers. Finally, for the *replication to additional hop* operation, we compare with the naive approach without additional replication.

6.3 Experimental Results

We now present our experimental results. We first present results in the fail-free mode, and then present results in the failure mode.

6.3.1 Evaluating *insertSucc* In this section we quantify the overhead of our *insertSucc* when compared to the naive *insertSucc*. The performance metric used is the time to complete the operation; this time is averaged over all such operations in the system during the run of the experiment.

We vary two parameters that affect the performance of the operations. The first parameter is the length of the ring successor list. The longer the list, the farther *insertSucc* has to propagate information before it can complete. The second is the ring stabilization period. The longer the stabilization period, the slower information about joining peers propagates due to stabilization.

Figure 19 shows the effect of varying the ring successor list length. There are several aspects to note about this figure. First, the time for our *insertSucc* increases linearly with the successor list length, while the time for the naive *insertSucc* remains constant. This is to be expected because the naive *insertSucc* only contacts the successor, while our *insertSucc* propagates information to as many predecessors as the length of the successor list. Second, perhaps surprisingly, the rate of increase of the time for our *insertSucc* operation is very small; this can be attributed to the optimization discussed in Section 4.3.1, where we proactively contact predecessors instead of only relying on the stabilization. Finally, an encouraging result is that the cost of our *insertSucc* is of the same ball park as that of the naive *insertSucc*; this means that users do not pay too high a price for consistency.

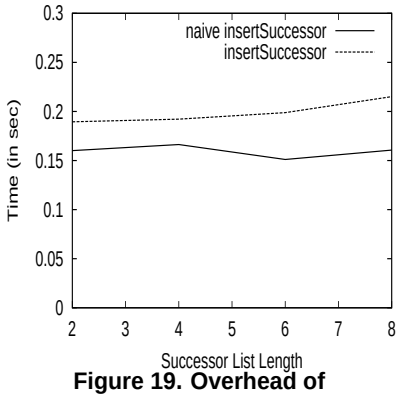

Figure 19. Overhead of insertSucc

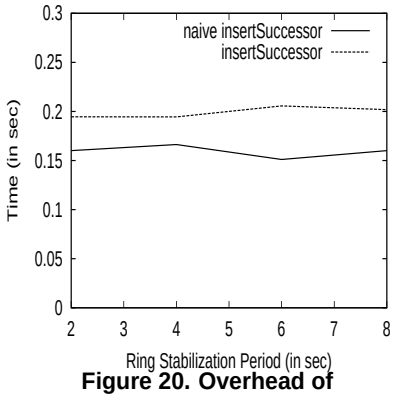

Figure 20. Overhead of insertSucc

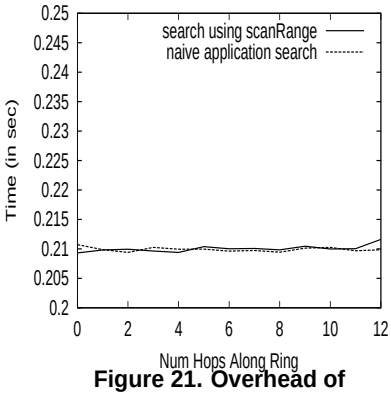

Figure 21. Overhead of scanRange

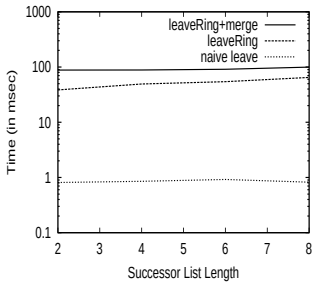

Figure 22. Overhead of leave

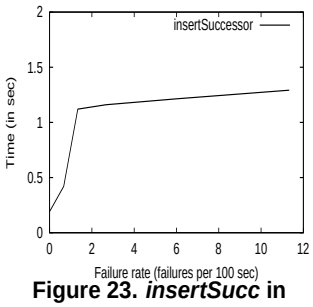

Figure 23. *insertSucc* in failure mode

Figure 20 shows the result of varying the ring stabilization frequency. The results are similar to varying the successor list length. Varying the ring stabilization period also has less of an effect on our *insertSucc* because of our optimization of proactively contacting predecessors.

6.3.2 Evaluating *scanRange* In this section, we investigate the overhead of using *scanRange* when compared to the naive approach of the application scanning the range by itself. Since the number of messages needed to complete the operation is the same for both approaches, we used the elapsed time to complete the range search as the relevant performance metric. We varied the size of the range to investigate its effect on performance, and averaged the elapsed time over all the searches requiring the same number of hops along the ring. Each peer generates searches for ranges of different sizes, and we measured the time needed to process the range search, once the first peer with items in the search range was found. This allows us to isolate the effects of scanning along the ring.

Figure 21 shows the performance results. As shown, there is practically no overhead to using *scanRange* as compared with the application level search; again, this indicates that the price of consistency is low. To our surprise, the time needed to complete the range search, for either approach, does not increase significantly with the increased number of hops. On further investigation, we determined that this was due to our experiments running on a cluster in the local area network. In a wide area network, we expect the time to complete a range search to increase significantly with the number of hops.

6.3.3 Evaluating *leave* and *Replicate to additional hop* In this section, we investigate the overhead of the proposed *leave* and *replicate to additional hop* operations as compared to the naive approach of simply leaving the ring without contacting any peer. For this experiment, we start with a system of 30 peers and delete items from the system that cause peers to merge and leave the ring.

We measure the time elapsed for three operations: (1) the *leave* operation in the ring, and (2) the merge operation in the Data Store (which includes the time for *replicate to additional hop*), and (3) the naive *leave*. Figure 22 shows the variation of the three times with successor list length. Note the log scale on y-axis. We observe that the *leave* and merge operations take approximately 100 msec, and do not constitute a big overhead. The naive version takes only 1 msec since it simply leaves the system.

6.3.4 Evaluation in Failure Mode We have so far studied the overhead of our proposed techniques in a system without failures. We now look at how our system behaves in a system with failures. In particular, we measure the variation of the average time taken for an insertSucc operation with the failure rate of peers. The system setting is as follows: We insert one peer every three seconds into the system, and we insert two items every second. We use the default successor list length (4) and default ring stabilization period (4 sec).

Figure 23 shows the variation of average time taken for a insertSucc operation with the peer failure rate. We observe that even in the case when the failure rate is as high as 1 in every 10 seconds, the time taken for insertSucc is not prohibitive (about 1.2 seconds compared to 0.2 seconds in a stable system).

7 Related Work

There has been a flurry of recent activity on developing indices for structured P2P systems. Some of these indices can efficiently support equality queries (e.g., [27, 31, 28]), while others can support both equality and range queries (e.g., [1, 2, 5, 6, 10, 12, 14, 15, 30]. This paper addresses query correctness and availability issues for such indices, which have not been previously addressed for range queries. Besides structured P2P indices, there are unstructured P2P indices such as [8, 13]. Unstructured in-

dices are robust to failures, but do not provide guarantees on query correctness and item availability. Since one of our main goals was to study correctness and availability issues, we focus on structured P2P indices.

There is a rich body of work on developing distributed index structures for databases (e.g., [18, 19, 21, 22, 23]. However, most of these techniques maintain consistency among the distributed replicas by using a *primary copy*, which creates both scalability and availability problems when dealing with thousands of peers. Some index structures, however, do maintain replicas lazily (e.g., [19, 21, 23]). However, these schemes are not designed to work in the presence of peer failures, dynamic item replication and reorganization, which makes them inadequate in a P2P setting. In contrast, our techniques are designed to handle peer failures while still providing correctness and availability guarantees.

Besides indexing, there is also some recent work on other data management issues in P2P systems such as complex queries [11, 16, 25, 26, 32, 33]. A correctness condition for processing aggregate queries in a dynamic network was proposed in [3]. An interesting direction for future work is to extend our techniques for query correctness and system availability to work for other complex queries such as keyword searches and joins.

8 Conclusion

We have introduced the first set of techniques that provably guarantee query correctness and system and item availability for range index structures in P2P systems. Our techniques provide provable guarantees, and they allow applications to abstract away all possible concurrency and availability issues. We have implemented our techniques in a real distributed P2P system, and quantified their performance.

As a next step, we would like to extend our approach to handle more complex queries such as joins and keyword searches.

9 Acknowledgements

This work was supported by NSF Grants CRCD-0203449, ITR-0205452, IIS- 0330201, and by AFOSR MURI Grant F49620-02-1-0233. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

References

[1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, 2001.

[2] J. Aspnes and G. Shah. Skip graphs. In *SODA*, 2003.

[3] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD*, 2004.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[5] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proc. SIGCOMM*, 2004.

[6] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: An index structure for peer-to-peer systems. In *Cornell Technical Report*, 2004.

[7] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. A storage and indexing framework for p2p systems. In *WWW Poster*, 2004.

[8] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer networks. In *ICDCS*, 2002.

[9] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[10] A. Daskos, S. Ghandeharizadeh, and X. An. Peper: A distributed range addressing space for p2p systems. In *DBISP2P*, 2003.

[11] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[12] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[13] Gnutella - http://gnutella.wego.com.

[14] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

[15] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[16] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, 2003.

[17] Jbi home page - http://www.rl.af.mil/programs/jbi/.

[18] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *IPPS*, 1992.

[19] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *SIGMOD*, 1993.

[20] D. Kossmann. The state of the art in distributed query processing. In *ACM Computing Surveys*, Sep 2000.

[21] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *SIGMOD*, 1994.

[22] W. Litwin, M.-A. Neimat, and D. Schneider. Rp*: A family of order preserving scalable distributed data structures. In *VLDB*, 1994.

[23] D. Lomet. Replicated indexes for distributed data. In *PDIS*, 1996.

[24] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1997.

[25] W. Ng, B. Ooi, K. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, 2003.

[26] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[27] S. Ratnasamy, M. H. P. Francis, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[29] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[30] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.

[31] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[32] I. Tatarinov and A. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD*, 2004.

[33] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards high performance peer-to-peer content and resource sharing systems. In *CIDR*, 2003.

APPENDIX

10 Preliminaries

In this section we introduce some useful terminology.

10.1 History

Definition 1 (History H) *Given a set of objects O, $\mathcal{H} = (O, <)$ is a* history *iff $<$ is a partial order defined on the objects in O.*

A *history H* of operations is a pair with first element a set of objects and the second element a partial order on these objects. We refer to the objects in the set O as *operations*. The partial order $<$ defines a *happened before* relationship among operations. Let $op_1, op_2 \in O$ be two operations in history H. If $op_1 < op_2$, then intuitively, op_1 has to finish before op_2 starts i.e op_1 *happened before* op_2. If op_1 and op_2 are not related by the partial order, then op_1 and op_2 could be executed in parallel.

We now define a projection of a *history* on a set of operations. We then define the notion of *ordered* operations.

Definition 2 (Projection History $\Pi_O(\mathcal{H})$) *Given a history $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, the* projection *of $\mathcal{H}$ with respect to a set of operations O (denoted by $\Pi_O(\mathcal{H})$) where $O \subseteq O_{\mathcal{H}}$ is the history $(O, <)$ where*

- *$x < y$ iff $x, y \in O \wedge x \leq_{\mathcal{H}} y$.*

Definition 3 (Ordered operations) *Given a history $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, let o_1, o_2 be two operations in $O_{\mathcal{H}}$. o_1 and o_2 are said to be* ordered *wrt each other in $\mathcal{H}$ iff $(o_1 \leq_{\mathcal{H}} o_2) \vee (o_2 \leq_{\mathcal{H}} o_1)$.*

Intuitively, two operations o_1 and o_2 are said to be *ordered* wrt to each other in a given history $\mathcal{H}$ iff one operation cannot be executed during the execution of the other.

10.2 Ring

In this section we define the notion of a *ring*. We define then an *abstract ring history* over the set of basic operations that *induce* a ring. We then specify the API we support on the ring. This API introduces a set of operations. We define an *API ring history* over these operations and hence a *ring* defined by this history.

Definition 4 (Ring) *Given a set of peers $\mathcal{P}$ and a bijection $\texttt{succ} : \mathcal{P} \rightarrow \mathcal{P}$, $\mathcal{R} = (\mathcal{P}, \texttt{succ})$ is a* ring *iff*

- *For any two peers $p, p' \in \mathcal{P}, p' = \texttt{succ}^k(p)$ for some $k \in \mathcal{N}$.*

A ring $\mathcal{R}$ is set of peers $\mathcal{P}$ with a successor function $succ$ defined on this set of peers. The successor function has the property that every peer is reachable from every other peer.

10.3 Abstract Ring History

Given a set of peers $\mathcal{P}$, the set of operations which define the ring are $O(\mathcal{P}) = \{insert(p, p'), leave(p), fail(p) | p, p' \in \mathcal{P}\}$. An *abstract ring history* on these operations is defined as follows:

Definition 5 (Abstract Ring History $\mathcal{H}$) *Given a set of peers $\mathcal{P}$, $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ is an* abstract ring history *iff*

1. *$\mathcal{H}$ is a history.*
2. $O_{\mathcal{H}} \subseteq O(\mathcal{P})$
3. $\exists p \in \mathcal{P}$ ($insert(p, p) \in O_{\mathcal{H}} \wedge (\forall p' \in \mathcal{P}\ insert(p', p') \in O_{\mathcal{H}} \Rightarrow p = p')$).

   (There exists a unique peer p which starts off the ring.)
4. $\forall p, p' \in \mathcal{P}$ ($insert(p, p') \in O_{\mathcal{H}} \Rightarrow \exists p''$ ($insert(p'', p) \in O_{\mathcal{H}} \wedge insert(p'', p) \leq_{\mathcal{H}} insert(p, p')$))

   (For every insert operation $insert(p, p')$ in the abstract ring history $\mathcal{H}$, p must be already part of the ring.)
5. $\forall p, p' \in \mathcal{P}, p \neq p'$, ($insert(p, p') \in O_{\mathcal{H}} \Rightarrow (insert(p', p') \notin O_{\mathcal{H}}) \wedge (\forall p'' \in \mathcal{P}\ insert(p'', p') \in O_{\mathcal{H}} \Rightarrow p = p'')$).

   (Any peer p' is inserted into the ring at most once).
6. $\forall p, p', p'' \in \mathcal{P}$ ($insert(p, p'), insert(p, p'') \in O_{\mathcal{H}} \Rightarrow (insert(p, p') \leq_{\mathcal{H}} insert(p, p'')) \vee (insert(p, p'') \leq_{\mathcal{H}} insert(p, p'))$).

   (Two ins operations in $\mathcal{H}$ on a given peer p cannot happen simultaneously.)
7. $\forall p \in \mathcal{P}$ ($fail(p) \notin O_{\mathcal{H}} \vee leave(p) \notin O_{\mathcal{H}}$).

   (At most one of $fail(p)$ or $leave(p)$ occurs in $\mathcal{H}$.)
8. $\forall p \in \mathcal{P}$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow (\forall p' \in \mathcal{P}\ insert(p, p') \in O_{\mathcal{H}} \Rightarrow insert(p, p') \leq_{\mathcal{H}} fail(p)) \wedge \exists p'$ ($insert(p', p) \in O_{\mathcal{H}} \wedge insert(p', p) \leq_{\mathcal{H}} fail(p)$)).

   (An insert operation involving p happened before *its failure and p is inserted* before *it fails.)*
9. $\forall p \in \mathcal{P}$ ($leave(p) \in O_{\mathcal{H}} \Rightarrow (\forall p' \in \mathcal{P}\ insert(p, p') \in O_{\mathcal{H}} \Rightarrow insert(p, p') \leq_{\mathcal{H}} leave(p)) \wedge \exists p'$ ($insert(p', p) \in O_{\mathcal{H}} \wedge insert(p', p) \leq_{\mathcal{H}} leave(p)$)).

   (An insert operation involving p happened before *a leave operation on p and p is inserted* before *the leave operation.)*

Note that according to the above definition of *abstract ring history* (or simply a *ring history*), we assume that a peer p once out of the ring (because of $fail$ or $leave$ operations), does not reenter with the same identifier. We are also not interested in failure of peers which are not part of the ring.

The following two claims will be useful later on.

Claim 1 *Given a ring history $\mathcal{H}$, $o \in O_{\mathcal{H}}, insert(p_0, p_0) \in O_{\mathcal{H}} \Rightarrow insert(p_0, p_0) \leq_{\mathcal{H}} o$.*

Proof: By definition of ring history, using condition 3, $\exists$ unique $p_0 \in \mathcal{P}$ $insert(p_0, p_0) \in O_{\mathcal{H}}$.

Let us consider possible options for operation o.

- $o = insert(p, p')$ for some $p, p' \in \mathcal{P}$

  If $p = p'$, from condition 3, $p = p' = p_0$. From the reflexivity of $<_{\mathcal{H}}$, $insert(p_0, p_0) <_{\mathcal{H}} insert(p_0, p_0)$.

  If $p \neq p'$, from condition 4, $\exists p''$ ($insert(p'', p) \in O_{\mathcal{H}} \wedge insert(p'', p) <_{\mathcal{H}} insert(p, p')$).

  If $p'' \neq p_0$, recursively using the same argument on $insert(p'', p)$, and using condition 5 to note that any peer is inserted into the ring at most once, we can conclude that $insert(p_0, p_0) <_{\mathcal{H}} \cdots <_{\mathcal{H}} insert(p'', p) <_{\mathcal{H}} insert(p, p')$. From transitivity of $<_{\mathcal{H}}$, $insert(p_0, p_0) <_{\mathcal{H}} insert(p, p')$.

  Therefore, $\forall p, p' \in \mathcal{P}$ $insert(p, p') \in O_{\mathcal{H}} \Rightarrow insert(p_0, p_0) <_{\mathcal{H}} insert(p, p')$.

- $o = fail(p)$

  Using condition 8, $\exists p'$ $insert(p', p) \in O_{\mathcal{H}} \wedge insert(p', p) <_{\mathcal{H}} fail(p)$.

  If $p' = p$ then $p' = p = p_0$ and therefore $insert(p_0, p_0) = insert(p, p) <_{\mathcal{H}} fail(p)$.

  Otherwise, $\exists p', p' \neq p$, $insert(p', p) \in O_{\mathcal{H}} \wedge insert(p', p) <_{\mathcal{H}} fail(p)$ and from the previous case $insert(p_0, p_0) <_{\mathcal{H}} insert(p', p)$. Therefore, from transitivity of $<_{\mathcal{H}}$, $insert(p_0, p_0) <_{\mathcal{H}} fail(p)$.

  Therefore $\forall p \in \mathcal{P} fail(p) \in O_{\mathcal{H}} \Rightarrow insert(p_0, p_0) <_{\mathcal{H}} fail(p)$.

- $o = leave(p)$

  Using condition 9, and same reasoning as for the above case we can conclude that $\forall p \in \mathcal{P}$ $leave(p) \in O_{\mathcal{H}} \Rightarrow insert(p_0, p_0) <_{\mathcal{H}} leave(p)$.

Therefore, $\forall o \in O_{\mathcal{H}}$, $insert(p_0, p_0) \in O_{\mathcal{H}} \Rightarrow insert(p_0, p_0) <_{\mathcal{H}} o$. ■

Claim 2 *Given a ring history $\mathcal{H}$, let $o \in O_{\mathcal{H}}$, $o \neq insert(p, p)$, $p \in \mathcal{P}$ such that $\nexists o' \in O_{\mathcal{H}}$ ($o <_{\mathcal{H}} o' \wedge o' \neq o$). History $\mathcal{H}' = (O_{\mathcal{H}} - \{o\}, <_{\mathcal{H}'})$, where $o_1 <_{\mathcal{H}'} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} - \{o\} \wedge o_1 <_{\mathcal{H}} o_2$, is also a ring history.*

Proof: We can prove this lemma by considering different possibilities for operation o: $insert(p, p'), fail(p), leave(p)$, and verifying that $\mathcal{H}'$ is a ring history in all these cases. ■

10.4 Ring $R_{\mathcal{H}}$ induced by history $\mathcal{H}$

Before we define an *induced ring*, we define the notion of *live* peers, given a *ring history $\mathcal{H}$*.

Definition 6 (Live peer) *Given a* ring history *$\mathcal{H}$, peer $p \in \mathcal{P}$ is said to be* live *in history $\mathcal{H}$, denoted by $live_{\mathcal{H}}(p)$, iff*

- $\exists p'$ $(insert(p', p) \in O_{\mathcal{H}} \wedge (fail(p), leave(p) \notin O_{\mathcal{H}})$).

A peer p is said to be *live* if it were inserted at some point in the ring history and did not fail or leave the ring at any point in the history. Given a *ring history $\mathcal{H}$*, we define the set of live peers, $\mathcal{P}_{\mathcal{H}} = \{p \in \mathcal{P} | live_{\mathcal{H}}(p)\}$.

Given a ring history $\mathcal{H}$, we define an *induced ring* $\mathcal{R}_{\mathcal{H}}$ as a pair with first element the set of live peers in $\mathcal{H}$ and second element a bijective successor function $succ_{\mathcal{H}}$ as follows:

Definition 7 (Induced Ring) *Given a* ring history *$\mathcal{H}$,* induced ring *$R_{\mathcal{H}} = (\mathcal{P}_{\mathcal{H}}, succ_{\mathcal{H}} : \mathcal{P}_{\mathcal{H}} \rightarrow \mathcal{P}_{\mathcal{H}})$ where $succ_{\mathcal{H}}$ is a bijective successor function defined by inducting on the number of operations in $O_{\mathcal{H}}$ as follows:*

- *Base step: $\mathcal{H} = (\{insert(p_0, p_0)\}, \{(p_0, p_0)\})$, for some peer $p_0 \in \mathcal{P}$. Define*

$$succ_{\mathcal{H}}(p_0) = p_0$$

  $\forall p \in \mathcal{P}, p \in \mathcal{P}_{\mathcal{H}} \Rightarrow p = p_0$. Therefore, $succ_{\mathcal{H}}$ is trivially bijective.

- *Induction hypothesis: For all $\mathcal{H}'$ such that $|O_{\mathcal{H}'}| = k, k \in \mathcal{N}^*$, assume that bijective successor function $succ_{\mathcal{H}'}$ is defined.*

- *Induction step: Consider a ring history $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, $|O_{\mathcal{H}}| = (k+1)$. Let $o \in O_{\mathcal{H}}$ such that $\nexists o' \in \mathcal{H}$ $o \leq_{\mathcal{H}} o' \wedge o' \neq o$. Such operation o exists, since H is finite. From Claim 2, $\mathcal{H}' = (O_{\mathcal{H}} - \{o\}, \leq_{\mathcal{H}'})$, where $o_1 \leq_{\mathcal{H}'} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} - \{o\} \wedge o_1 \leq_{\mathcal{H}} o_2$, is a* ring history. *Since $|\mathcal{H}'| = k$, by induction hypothesis, $\mathcal{H}'$ has a bijective successor function, $succ_{\mathcal{H}'} : P_{\mathcal{H}'} \rightarrow P_{\mathcal{H}'}$.*

  From Claim 1 o cannot be $insert(p, p)$ for some $p \in \mathcal{P}_{\mathcal{H}}$. Let us consider the possible options for o:

  - *$o = insert(p', p'')$ for some $p' \in \mathcal{P}_{\mathcal{H}'}$ and $p'' \in \mathcal{P}$, $p'' \notin \mathcal{P}_{\mathcal{H}'}$. Note that $\mathcal{P}_{\mathcal{H}} = \mathcal{P}_{\mathcal{H}'} \cup p''$. Define*

$$succ_{\mathcal{H}}(p) = \begin{cases} succ_{\mathcal{H}'}(p) & p \in \mathcal{P}_{\mathcal{H}'}, p \neq p', p'' \\ p'' & p = p' \\ succ_{\mathcal{H}'}(p') & p = p'' \end{cases}$$

  - *$o = fail(p') \vee leave(p')$ for some $p' \in \mathcal{P}_{\mathcal{H}'}$. Note that $\mathcal{P}_{\mathcal{H}} = \mathcal{P}_{\mathcal{H}'} - \{p'\}$. Let $p'' \in \mathcal{P}_{\mathcal{H}'}$ such that $succ_{\mathcal{H}'}(p'') = p'$. Define*

$$succ_{\mathcal{H}}(p) = \begin{cases} succ_{\mathcal{H}'}(p) & p \in \mathcal{P}_{\mathcal{H}'}, p \neq p'' \\ succ_{\mathcal{H}'}(p') & p = p'' \end{cases}$$

  We can easily verify that $succ_{\mathcal{H}}$ is bijective in all of the above cases.

Given a ring history $\mathcal{H}$ and an *induced ring* $R_{\mathcal{H}} = (P_{\mathcal{H}}, succ_{\mathcal{H}} : P_{\mathcal{H}} \rightarrow P_{\mathcal{H}})$, we define sequence of operations $S_{\mathcal{H}}$ as the order in which operations were considered in defining $succ_{\mathcal{H}}$. Given a ring history $\mathcal{H}$, we now show that for any sequence allowed by the definition of $succ_{\mathcal{H}}$, the resulted successor function $succ_{\mathcal{H}}$ is unique.

Claim 3 *Given a ring history $\mathcal{H}$, $succ_{\mathcal{H}}$ as defined above is unique.*

Proof: We prove the following two claims before we proceed to prove the main claim.

Claim A: Given a ring history $\mathcal{H}$, let $o_1, o_2 \in O_{\mathcal{H}}$ be two non-*ordered* operations. Sequences $S_{\mathcal{H}} = S_1, o_1, o_2, S_2$ and $S'_{\mathcal{H}} = S_1, o_2, o_1, S_2$ define the same $succ_{\mathcal{H}}$.

Proof: Consider all possible o_1, o_2 such that $o_1, o_2 \in O_{\mathcal{H}}$ are two non-*ordered* operations. For each such pair we show that $S_{\mathcal{H}}$ and $S'_{\mathcal{H}}$ yields the same $succ_{\mathcal{H}}$.

- $o_1 = insert(p, p)$ for some $p \in \mathcal{P}$: Note that $\not\exists o_2$ o_1, o_2 are non-ordered. Therefore, given non-ordered operations o_1, o_2, neither o_1 nor o_2 is $insert(p, p)$ for some $p \in \mathcal{P}$.
- $o_1 = insert(p, p'), p \neq p'$: $fail(p)$ or $fail(p')$ are ordered wrt o_1 and so are $leave(p)$ or $leave(p')$. Also $insert(p'', p''')$ is ordered wrt o_1 if one of p'' or p''' equals p or p'. Therefore, only possibilities for non-ordered operations when $o_1 = insert(p, p')$ are:

  -$o_2 = insert(p'', p'''), (p'' \neq p'''$ and $p'', p''' \neq p, p')$: After considering the sequence of operations in S_1, let successor function be $succ_{S_1}$.

  Let $S_{11} = S_1, o_1$. By definition of $succ_{\mathcal{H}}$,

$$succ_{S_{11}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p' \\ p' & p_1 = p \\ succ_{S_1}(p) & p_1 = p' \end{cases}$$

  Now let $S_{12} = S_1, o_1, o_2$. By definition of $succ_{\mathcal{H}}$,

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_{11}}(p_1) & p_1 \in \mathcal{P}_{S_{11}}, p_1 \neq p'', p''' \\ p''' & p_1 = p'' \\ succ_{S_{11}}(p'') & p_1 = p''' \end{cases}$$

  Therefore,

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p', p'', p''' \\ p''' & p_1 = p'' \\ p' & p_1 = p \\ succ_{S_1}(p'') & p_1 = p''' \\ succ_{S_1}(p) & p_1 = p' \end{cases}$$

  We can similarly see that for $S'_{21} = S_1, o_2, o_1$,

$$succ_{S'_{21}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p', p'', p''' \\ p''' & p_1 = p'' \\ p' & p_1 = p \\ succ_{S_1}(p'') & p_1 = p''' \\ succ_{S_1}(p) & p_1 = p' \end{cases}$$

  Therefore, $succ_{S_{12}} = succ_{S'_{21}}$ and hence $succ_{S_{\mathcal{H}}} = succ_{S'_{\mathcal{H}}}$.

  -$o_2 = fail(p'') \vee leave(p''), (p'' \neq p, p')$: $succ_{S_{11}}$ is same as in the previous case.

  Now let $S_{12} = S_1, o_1, o_2$. Let p''' be such that $succ_{S_{11}}(p''') = p''$. By definition of $succ_{\mathcal{H}}$,

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_{11}}(p_1) & p_1 \in \mathcal{P}_{S_{11}}, p_1 \neq p''' \\ succ_{S_{11}}(p'') & p_1 = p''' \end{cases}$$

  If $p''' = p'$, then

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p' \\ p' & p_1 = p \\ succ_{S_1}(p'') & p_1 = p' \end{cases}$$

  If $p''' \neq p'$, then

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p', p''' \\ p' & p_1 = p \\ succ_{S_1}(p) & p_1 = p' \\ succ_{S_1}(p'') & p_1 = p''' \end{cases}$$

  We can similarly see that for $S'_{21} = S_1, o_2, o_1$:

  If $p''' = p'$, then

$$succ_{S'_{21}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p' \\ p' & p_1 = p \\ succ_{S_1}(p'') & p_1 = p' \end{cases}$$

  If $p''' \neq p'$, then

$$succ_{S'_{21}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p, p', p''' \\ p' & p_1 = p \\ succ_{S_1}(p) & p_1 = p' \\ succ_{S_1}(p'') & p_1 = p''' \end{cases}$$

  Therefore, in either case, $succ_{S'_{21}} = succ_{S_{12}}$.

- $o_1 = fail(p) \vee leave(p)$

  -$o_2 = fail(p') \vee leave(p'), (p' \neq p)$:

  Let $S_{11} = S_1, o_1$. Let p'' be such that $succ_{S_1}(p'') = p$. By definition of $succ_{\mathcal{H}}$,

$$succ_{S_{11}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p'' \\ succ_{S_1}(p) & p_1 = p'' \end{cases}$$

  Now let $S_{12} = S_1, o_1, o_2$. Let p''' be such that $succ_{\mathcal{H},S_{11}}(p''') = p'$. By definition of $succ_{\mathcal{H}}$,

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_{11}}(p_1) & p_1 \in \mathcal{P}_{S_{11}}, p_1 \neq p''' \\ succ_{S_{11}}(p') & p_1 = p''' \end{cases}$$

  If $(p' \neq p'') \wedge (p''' \neq p'')$,

$$succ_{S_{12}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p'', p''' \\ succ_{S_1}(p) & p_1 = p'' \\ succ_{S_1}(p') & p_1 = p''' \end{cases}$$

  We can similarly see that for $S'_{21} = S_1, o_2, o_1$, if $(p' \neq p'') \wedge (p''' \neq p'')$,

$$succ_{S'_{21}}(p_1) = \begin{cases} succ_{S_1}(p_1) & p_1 \in \mathcal{P}_{S_1}, p_1 \neq p'', p''' \\ succ_{S_1}(p) & p_1 = p'' \\ succ_{S_1}(p') & p_1 = p''' \end{cases}$$

  Other cases are similar.

  -$o_2 = insert(p', p''), (p', p'' \neq p)$: This case is same as the case above with $o_1 = insert(p, p')$ and $o_2 = fail(p'') \vee leave(p''), (p'' \neq p, p')$.

■

Claim B: Given a history $\mathcal{H}$ and sequences $S_{\mathcal{H}}, S'_{\mathcal{H}}$, let $S'_{\mathcal{H}}$ be obtained from $S_{\mathcal{H}}$ by swapping consecutive non-*ordered* operations. Sequences $S_{\mathcal{H}}$ and $S'_{\mathcal{H}}$ define the same $succ_{\mathcal{H}}$.

Proof: For every swap of non-*conflicting* operations, use Claim A. ■

Proof of the main claim: By induction on number of operations in $O_{\mathcal{H}}$.

Base step: $\mathcal{H} = (\{insert(p_0, p_0)\}, \{(p_0, p_0)\})$, for some peer $p_0 \in \mathcal{P}$. In this case there is only one possible sequence $S = \{insert(p_0, p_0)\}$ and hence $succ_{\mathcal{H}} = \{(p_0, p_0)\}$ is unique.

Induction hypothesis: Assume for all $\mathcal{H}$ such that $|O_{\mathcal{H}}| = k, k \in \mathcal{N}^*$, $succ_{\mathcal{H}}$ is unique. In other words, any two valid sequence $S_{\mathcal{H}}$ and $S'_{\mathcal{H}}$ define the same successor function i.e $succ_{S_{\mathcal{H}}} = succ_{S'_{\mathcal{H}}}$.

Induction step: Consider a ring history $\mathcal{H}$ such that $|O_{\mathcal{H}}| = (k+1), k > 0$. Consider any two valid sequences $S_1 = S_{11}, o_1$ and $S_2 = S_{21}, o_2$, where $o_1, o_2 \in O_{\mathcal{H}}$ such that $\nexists o' \in \mathcal{H}\ (o_1 \leq_{\mathcal{H}} o') \vee (o_2 \leq_{\mathcal{H}} o')$.

If $o_1 = o_2 = o$, using Claim 2, $\mathcal{H}' = (O_{\mathcal{H}} - \{o\}, \leq_{\mathcal{H}'})$, where $o_1 \leq_{\mathcal{H}'} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} - \{o\} \wedge o_1 \leq_{\mathcal{H}} o_2$, is a *ring history*. Since $|\mathcal{H}'| = k$, using the induction hypothesis, sequences S_{11} and S_{21} yield the same successor function $succ_{\mathcal{H}'}$. Therefore, in this case, $succ_{\mathcal{H}}$ is unique.

If $o_1 \neq o_2$: define $\mathcal{H}'_1 = (O_{\mathcal{H}} - \{o_1\}, \leq_{\mathcal{H}'_1})$, where $o \leq_{\mathcal{H}'_1} o'$ iff $o, o' \in O_{\mathcal{H}} - \{o_1\} \wedge o \leq_{\mathcal{H}} o'$, and $\mathcal{H}'_2 = (O_{\mathcal{H}} - \{o_2\}, \leq_{\mathcal{H}'_2})$, where $o \leq_{\mathcal{H}'_2} o'$ iff $o, o' \in O_{\mathcal{H}} - \{o_1\} \wedge o \leq_{\mathcal{H}'} o'$. From Claim 2, $\mathcal{H}'_1$ and $\mathcal{H}'_2$ are *ring histories*.

Now consider sequence S_1. $o_2 \in S_1$. Since $\nexists o' \in \mathcal{H}\ (o_1 \leq_{\mathcal{H}} o') \vee (o_2 \leq_{\mathcal{H}} o')$., for all operations o after o_2, it cannot happen that $o \leq_{\mathcal{H}} o_2$.

Therefore, o_2 is not *ordered* wrt to all the operations after o_2 in S_1. Using Claim B above, considering the new sequence $S'_1 = S'_{11}, o_2$ in which o_2 is swapped out and moved all the way to the end, $succ_{S_1} = succ_{S'_1}$.

Now since $\mathcal{H}'_2$ is a ring history, using the induction hypothesis, $succ_{S'_{11}} = succ_{S_{21}}$ and therefore, $succ_{S'_1} = succ_{S_2}$. We therefore have, $succ_{S_1} = succ_{S_2}$.

Hence proved. ■

Lemma 1 (Induced Ring) *Given a ring history $\mathcal{H}$, the* induced ring $R_{\mathcal{H}}$ *defined above is a* ring*, as defined in Definition 4.*

Proof: By induction on the number of operations in $O_{\mathcal{H}}$.

Base step When $\mathcal{H} = (\{insert(p_0, p_0)\}, \{(p_0, p_0)\})$ for some peer $p_0 \in \mathcal{P}$, trivially $R_{\mathcal{H}}$ is a ring.

Induction hypothesis: Assume that for all $\mathcal{H}'$ such that $|O_{\mathcal{H}'}| = m$, for any two peers $p, q \in \mathcal{P}_{\mathcal{H}'}, q = succ^k_{\mathcal{H}}(p)$ for some $k \in \mathcal{N}$.

Induction step: Consider a ring history $\mathcal{H}$. Let $|O_{\mathcal{H}}| = (m+1)$. Consider an op $o \in O_{\mathcal{H}}$ for which $\nexists o' \in O_{\mathcal{H}}, o' \neq o \wedge o \leq_{\mathcal{H}} o'$. From Claim 2, $(O_{\mathcal{H}'}, \leq_{\mathcal{H}'}) = (O_{\mathcal{H}} - \{o\}, \leq_{\mathcal{H}'})$, where $o_1 \leq_{\mathcal{H}'} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} - \{o\} \wedge o_1 \leq_{\mathcal{H}} o_2$, is a *ring history*. Since $|O_{\mathcal{H}'}| = m$, by induction hypothesis, for any two peers $p, q \in \mathcal{P}_{\mathcal{H}'}, q = succ^k_{\mathcal{H}}(p)$ for some $k \in \mathcal{N}$.

Now, let us consider the possible operations for o.

- $o = insert(p', p'')$ for some $p' \in \mathcal{P}_{\mathcal{H}'}$ and $p'' \in \mathcal{P}$, $p'' \notin \mathcal{P}_{\mathcal{H}'}$.

  Note that $\mathcal{P}_{\mathcal{H}} = \mathcal{P}_{\mathcal{H}'} \cup p''$. By induction hypothesis, for any two peers $p, q \in \mathcal{P}_{\mathcal{H}'}, q = succ^k_{\mathcal{H}}(p)$ for some $k \in \mathcal{N}$.

  Given two peers $p, q \in \mathcal{P}_{\mathcal{H}'}$. Let $q = succ^k_{\mathcal{H}'}(p)$ and $p' = succ^{k'}_{\mathcal{H}'}(p)$.

  Now let us assume $k \leq k'$. By definition of $succ_{\mathcal{H}}$, $succ_{\mathcal{H}}(p) = succ_{\mathcal{H}'}(p) \forall p \neq p', p'' \in \mathcal{P}_{\mathcal{H}}$. Since $q = succ^k_{\mathcal{H}'}(p)$, $p' = succ^{k'}_{\mathcal{H}'}(p)$, $k \leq k'$ we can conclude that $q = succ^k_{\mathcal{H}}(p)$.

Now let us assume $k > k'$. $p' = succ^{k'}_{\mathcal{H}'}(p)$, $q = succ^{k}_{\mathcal{H}'}(p)$ and therefore, $q = succ^{(k-k'-1)}_{\mathcal{H}'}(succ_{\mathcal{H}'}(p'))$

By definition of $succ_{\mathcal{H}}$, $succ_{\mathcal{H}}(p) = succ_{\mathcal{H}'}(p)$ $\forall p \neq p', p'' \in \mathcal{P}_{\mathcal{H}}$ and therefore $p' = succ^{k'}_{H}(p)$ and $q = succ^{(k-k'-1)}_{H}(succ_{\mathcal{H}'}(p'))$. Again, by definition of $succ_{\mathcal{H}}$, $succ_{\mathcal{H}'}(p') = succ_{\mathcal{H}}(p'')$ and $succ_{\mathcal{H}}(p') = p''$.

Therefore, $q = succ^{(k-k'-1)}_{H}(succ_{\mathcal{H}'}(p'))$ = $succ^{(k-k'-1)}_{H}(succ_{\mathcal{H}}(p''))$ = $succ^{(k-k'+1)}_{H}(p')$ = $succ^{(k+1)}_{\mathcal{H}}(p)$.

Therefore, for any two peers $p, q \in \mathcal{P}_{\mathcal{H}}$, $q = succ^{k''}_{\mathcal{H}}(p)$ for some $k'' \in \mathcal{N}$.

- $o = fail(p') \vee leaveRing(p')$ for some $p' \in \mathcal{P}_{\mathcal{H}'}$.

  Note that $\mathcal{P}_{\mathcal{H}} = \mathcal{P}_{\mathcal{H}'} - p'$. By induction hypothesis, for any two peers $p, q \in \mathcal{P}_{\mathcal{H}'}, q = succ^{k}_{\mathcal{H}}(p)$ for some $k \in \mathcal{N}$.

  Given two peers $p, q \neq p' \in \mathcal{P}_{\mathcal{H}'}$. Let $p'' \in \mathcal{P}_{\mathcal{H}} \ni succ(p'') = p'$. Let $q = succ^{k}_{\mathcal{H}'}(p)$ and $p'' = succ^{k'-1}_{\mathcal{H}'}(p), k \neq k'$. Then $p' = succ^{k'}_{\mathcal{H}'}(p)$

  Now let us assume $k < k'$. By definition of $succ_{\mathcal{H}}$, $succ_{\mathcal{H}}(p) = succ_{\mathcal{H}'}(p) \forall p \neq p' \in \mathcal{P}_{\mathcal{H}}$. Since $q = succ^{k}_{\mathcal{H}'}(p)$, $p' = succ^{k'}_{\mathcal{H}'}(p)$, $k < k'$ we can conclude that $q = succ^{k}_{H}(p)$.

  Now let us assume $k > k'$. $p' = succ^{k'}_{\mathcal{H}'}(p)$, $q = succ^{k}_{\mathcal{H}'}(p)$ and therefore, $q = succ^{(k-k'-1)}_{\mathcal{H}'}(succ_{\mathcal{H}'}(p'))$.

  By definition of $succ_{\mathcal{H}}$, $succ_{\mathcal{H}}(p) = succ_{\mathcal{H}'}(p)$ $\forall p \neq p' \in \mathcal{P}_{\mathcal{H}}$ and therefore $p' = succ^{k'}_{H}(p)$ and $q = succ^{(k-k'-1)}_{H}(succ_{\mathcal{H}'}(p'))$. Again, by definition of $succ_{\mathcal{H}}$, $succ_{\mathcal{H}'}(p')$ = $succ_{\mathcal{H}}(p'')$. So, $q = succ^{(k-k'-1)}_{H}(succ_{\mathcal{H}'}(p'))$ = $succ^{(k-k'-1)}_{H}(succ_{\mathcal{H}}(p''))$ = $succ^{(k-k')}_{H}(p'')$ = $succ^{(k-1)}_{\mathcal{H}}(p)$.

  Therefore, for any two peers $p, q \in \mathcal{P}_{\mathcal{H}}$, $q = succ^{k''}_{\mathcal{H}}(p)$ for some $k'' \in \mathcal{W}$, $k'' \leq |\mathcal{P}_{\mathcal{H}}|$.

■

11 Ring Correctness

11.1 API Ring History

To define the notion of an *API ring history* we need to first understand the operations required to specify the ring API. There are six methods exposed as part of the ring API:

- p.`initRing()`
- p.`initInsert(`p'`)`
- p.`initLeave()`
- p.`initGetSucc()`
- p.`initSendToSucc(`msg`)`
- p.`fail()`

The invocation of each of these API methods is associated with an operation. For example, $initLeave(p)$ is the operation associated with the invocation of the API method p.`initLeave()`. An API method need not return right away. An event is thrown to signal the end of the API call. For example, $leave(p)$ is the operation used to signal the end of the API call p.`initLeave()`, initiated at peer p.

We now run through the API methods, explain what the API method does and specify the init and end operations associated with the method.

- p.`initRing()`: This method inserts peer p as the first peer in the ring. $initRing(p)$ is the operation associated with the invocation of p.`initRing()`. $insert(p,p), inserted(p)$ are the operations used to signal the end of the API call p.`initRing()`, initiated at peer p.
- p.`initInsert(`p'`)`: This method invoked at peer p inserts peer p' into the ring as the successor of peer p. $initInsert(p,p')$ is the operation associated with the invocation of p.`initRing(`p'`)`. $insert(p,p')$ is the operation used to signal the end of the API call p.`initInsert(`p'`)`, initiated at peer p; $inserted(p')$ is the operation used to signal the insertion of peer p' into the ring at peer p'. Note that this is the same operation used to signal the end of the API method p.`initRing()`.
- p.`initLeave()`: This method causes peer p to *leave* the ring. $initLeave(p)$ is the operation associated with the invocation of p.`initLeave()`. $leave(p)$ is the operation used to signal the end of the API method p.`initLeave()`, initiated at peer p.
- p.`initGetSucc()`: This method returns p', the *current* successor of peer p. $initGetSucc_i(p), i \in \mathcal{N}$, is the operation associated with the invocation of p.`initGetSucc()`. Note here that subscript i is used to distinguish p.`initGetSucc()` API calls with the same signature. $getSucc_i(p,p'), i \in \mathcal{N}$, is the operation used to signal the end of the API method p.`initGetSucc()`, initiated at peer p.
- p.`initSendToSucc(`msg`)`: This method sends msg to the successor of p'. $initSendToSucc_i(p), i \in \mathcal{N}$ is the operation associated with the invocation of p.`initSendToSucc`. $sendToSucc_i(p,p'), i \in \mathcal{N}$ is the operation used to signal the end of the API method p.`initSendToSucc()`, initiated at peer p; $recvSendToSucc(p,p')$ is the operation associated with the completion of the API operation at peer p'.
- p.`fail()`: $p.fail()$ is included in the API to capture peer failures. It is not a method which can be called by higher layers. We use the operation $fail(p)$ to denote the failure of a peer p.

In addition, the following events are thrown up by the ring:

- `INFOFORSUCCEVENT`: This event is thrown by the ring before it contacts a new successor, during stabilization, so higher layers can send some information to the successor. $initInfoForSuccEvent_i(p)$ is the operation used to denote initiation of event at peer p. The ring waits for a response for this event. $infoForSuccEvent_i(p)$ indicates that the event has been handled by the higher layers at peer p.
- `INFOFROMPREDEVENT`: This event is thrown when some information is received from the predecessor. $initInfoFromPredEvent_i(p,p')$ is the operation used to denote the initiation of this event at peer p when some data is received from peer p. $infoFromPredEvent_i(p,p')$ is the operation used to denote the completion of this event.
- `NEWSUCCEVENT`: This event is thrown when a new successor is detected. The ring does not expect any response from higher layers for this event. ***newSuccEvent(p,p')*** denotes the new successor event at peer p with new successor p'.
- `INSERT`: This event is thrown when a new peer is to be inserted as successor. The operation $initInsertEvent_i(p,p')$ signals the initiation of this event at peer p. $insert_i(p,p')$ signals the end of the event and of the `p.initInsert(p')` API operation. The ring waits for the response from this event.
- `INSERTED`: This event is thrown up to signal that the peer is inserted into the ring. The ring does not wait for any response from higher layers. The operation associated is $inserted(p)$.
- `LEAVE`: This event is thrown up to signal the end of `p.initLeave()` API method. The associated operation is ***leave(p)***. The ring does not expect any response from higher layers for this event.

Given a set of peers $\mathcal{P}$, the set of operations under consideration are $\mathcal{O}(\mathcal{P})$ $=\{initRing(p), inserted(p), initInsert(p,p'), initInsertEvent(p,p'), insert(p,p'), initLeave(p), leave(p), initGetSucc_i(p), getSucc_i(p,p'), initSendToSucc_i(p), recvSendToSucc_i(p,p'), sendToSucc_i(p,p'), fail(p), initInfoForSuccEvent_i(p), infoForSuccEvent_i(p), initInfoFromPredEvent_i(p,p'), infoFromPredEvent_i(p,p'), newSuccEvent(p,p') | p,p' \in \mathcal{P}, i \in \mathcal{N}\}$.

Before we define an *API ring history*, we first present some useful definitions and notations.

Notation($\mathcal{O}(p)$): $\mathcal{O}(p) = \{initRing(p), inserted(p), initLeave(p), leave(p), initGetSucc_i(p), initSendToSucc_i(p), fail(p), initInfoForSuccEvent_i(p), infoForSuccEvent_i(p)\}$ denotes all operations at peer p involving only peer p.

Notation($\mathcal{O}(p,p')$): $\mathcal{O}(p,p') = \{initInsert(p,p'), initInsertEvent(p,p'), insert(p,p'), getSucc_i(p,p'), recvSendToSucc_i(p,p'), sendToSucc_i(p,p'), initInfoFromPredEvent_i(p,p'), infoFromPredEvent_i(p,p'), newSuccEvent(p,p')\}$ denotes all operations at peer p involving only peers p and p'.

Definition 8 (Truncated History $\mathcal{H}_o$) *Given a history $\mathcal{H} = (O_\mathcal{H}, \leq_\mathcal{H})$ and an operation $o \in O_\mathcal{H}$, $\mathcal{H}_o = (O_{\mathcal{H}_o}, \leq_{\mathcal{H}_o})$ is a* truncated history *iff*

- $O_{\mathcal{H}_o} = \{o' \in O_\mathcal{H} | o' \leq_\mathcal{H} o\}$.
- $o_1, o_2 \in O_{\mathcal{H}_o} \wedge o_1 \leq_\mathcal{H} o_2 \Rightarrow o_1 \leq_{\mathcal{H}_o} o_2$.

We now define an *API Ring History* as follows:

Definition 9 (API Ring History $\mathcal{H}$) *Given a set of peers $\mathcal{P}$, $\mathcal{H} = (O_\mathcal{H}, \leq_\mathcal{H})$ is an* API ring history *iff*

1. *$\mathcal{H}$ is a history.*
2. $O_\mathcal{H} \subseteq \mathcal{O}(\mathcal{P})$
3. API restrictions

   (a) $\exists p \in \mathcal{P}$ ($initRing(p) \in O_\mathcal{H} \wedge insert(p,p) \in O_\mathcal{H} \wedge (\forall p' \in \mathcal{P}\ initRing(p') \in O_\mathcal{H} \Rightarrow p = p')$).
   (There exists a unique peer p which starts off the ring.)

   (b) $\forall p, p' \in \mathcal{P}, p \neq p'$, ($initInsert(p,p') \in O_\mathcal{H} \Rightarrow (initRing(p') \notin O_\mathcal{H} \wedge (\forall p'' \in \mathcal{P}\ initInsert(p'',p') \Rightarrow p'' = p))$).
   (Insert of peer p' is tried at most once. This is not a necessary but a convenient API restriction.)

   (c) $\forall p \in \mathcal{P} (\forall op(p) \in \{initLeave(p), fail(p), initGetSucc_i(p), initSendToSucc_i(p), \forall i \in \mathcal{N}\} (op(p) \in O_\mathcal{H} \Rightarrow inserted(p) \in O_\mathcal{H} \wedge inserted(p) \leq_\mathcal{H} op(p)) \wedge (\forall p' \in \mathcal{P}, \forall op(p,p') \in \{initInsert(p,p')\} (op(p,p') \in O_\mathcal{H} \Rightarrow inserted(p) \in O_\mathcal{H} \wedge inserted(p) \leq_\mathcal{H} op(p,p'))))$
   (All operations on peer p except $initRing(p)$ are initiated after $inserted(p)$.)

   (d) $\forall p \in \mathcal{P} (fail(p) \in O_\mathcal{H} \Rightarrow (op(p) \in \{initRing(p), initLeave(p), initGetSucc_i(p), initSendToSucc_i(p), \forall i \in \mathcal{N}\} \wedge op(p) \in O_\mathcal{H} \Rightarrow op(p) \leq_\mathcal{H} fail(p)) \wedge (\forall p' \in \mathcal{P} (op(p,p') \in \{initInsert(p,p')\} \wedge op(p,p') \in O_\mathcal{H} \Rightarrow op(p,p') \leq_\mathcal{H} fail(p))))$
   (All operations on peer p are initiated before $fail(p)$.)

   (e) $\forall p \in \mathcal{P} (leave(p) \in O_\mathcal{H} \Rightarrow (op(p) \in \{initRing(p), initLeave(p), initGetSucc_i(p), initSendToSucc_i(p), \forall i \in \mathcal{N}\} \wedge op(p) \in O_\mathcal{H} \Rightarrow op(p) \leq_\mathcal{H} leave(p)) \wedge (\forall p' \in \mathcal{P} (op(p,p') \in \{initInsert(p,p')\} \wedge op(p,p') \in O_\mathcal{H} \Rightarrow op(p,p') \leq_\mathcal{H} leave(p))))$
   (All operations on peer p are initiated before $leave(p)$.)

   (f) $\forall p \in \mathcal{P}$ ($leave(p) \in O_\mathcal{H} \Rightarrow (fail(p) \in O_\mathcal{H} \Rightarrow fail(p) \leq_\mathcal{H} leave(p))$).
   (API fail operation on peer p cannot occur after $leave(p)$)

4. Semantic requirements

   (a) $\mathcal{H}_a = \Pi_{O(\mathcal{P})}(\mathcal{H})$ *is an* abstract ring history
   (Projection of an API ring history *on the operations used to define an* abstract ring history *should give us an* abstract ring history*).*

   (b) $\forall p \in \mathcal{P}$ ($fail(p) \in O_\mathcal{H} \Rightarrow (\forall op(p) \in \mathcal{O}(p)$ ($op(p) \in O_\mathcal{H} \wedge op(p) \neq fail(p) \wedge op(p) \neq leave(p) \Rightarrow op(p) \leq_\mathcal{H} fail(p)$)) $\wedge$ ($\forall p' \in \mathcal{P}, \forall op(p,p') \in \mathcal{O}(p,p')$ ($op(p,p') \in O_\mathcal{H} \Rightarrow op(p,p') \leq_\mathcal{H} fail(p)$)))
   (Any operation at p other than fail or leave happened before $fail(p)$*.)*

   (c) $\forall p \in \mathcal{P}$ ($leave(p) \in O_\mathcal{H} \Rightarrow (\forall op(p) \in \mathcal{O}(p)$ ($op(p) \in O_\mathcal{H} \wedge op(p) \neq fail(p) \wedge op(p) \neq leave(p) \Rightarrow op(p) \leq_\mathcal{H} leave(p)$)) $\wedge$ ($\forall p' \in \mathcal{P}, \forall op(p,p') \in \mathcal{O}(p,p')$ ($op(p,p') \in O_\mathcal{H} \Rightarrow op(p,p') \leq_\mathcal{H} leave(p)$)))
   (Any operation at p other than fail or leave happened before $leave(p)$*.)*

   (d) $\forall p \in \mathcal{P}$ ($insert(p,p) \in O_\mathcal{H} \Rightarrow initRing(p) \in O_\mathcal{H} \wedge initRing(p) \leq_\mathcal{H} insert(p,p)$).
   (An insert operation on first peer p is initiated before it is completed.)

   (e) $\forall p, p' \in \mathcal{P}, p \neq p'$, ($insert(p,p') \in O_\mathcal{H} \Rightarrow initInsert(p,p') \in O_\mathcal{H} \wedge initInsert(p,p') \leq_\mathcal{H} insert(p,p')$).

$\forall p, p' \in \mathcal{P}, p \neq p',\ (\ insert(p,p') \in O_{\mathcal{H}} \Rightarrow initInsertEvent(p,p') \in O_{\mathcal{H}} \wedge initInsertEvent(p,p') \leq_{\mathcal{H}} insert(p,p')\ )$.
(An insert operation on any peer p is initiated before it is completed.)

(f) $\forall p \in \mathcal{P}\ (\ inserted(p) \in O_{\mathcal{H}} \Rightarrow (initRing(p) \in O_{\mathcal{H}}) \vee (\ \exists p' \in \mathcal{P}\ initInsert(p',p) \in O_{\mathcal{H}} \wedge initInsert(p',p) \leq_{\mathcal{H}} inserted(p)\ )\ )$.
(An insert operation on any peer p is initiated before it is completed.)

(g) $\forall p \in \mathcal{P}\ (\ leave(p) \in O_{\mathcal{H}} \Rightarrow initLeave(p) \in O_{\mathcal{H}} \wedge initLeave(p) \leq_{\mathcal{H}} leave(p)\ )$.
(A leave operation on any peer p is initiated before it is completed.)

(h) $\forall p, p' \in \mathcal{P}\ (\forall i \in \mathcal{N}(\ getSucc_i(p,p') \in O_{\mathcal{H}}) \Rightarrow initGetSucc_i(p) \in O_{\mathcal{H}} \wedge initGetSucc_i(p) \leq_{\mathcal{H}} getSucc_i(p,p')\ )$.
(A $getSucc$ operation on any peer p is initiated before it is completed.)

(i) $\forall p, p' \in \mathcal{P}\ (\ \forall i \in \mathcal{N}(recvSendToSucc_i(p,p') \in O_{\mathcal{H}}) \Rightarrow initSendToSucc_i(p) \in O_{\mathcal{H}} \wedge initSendToSucc_i(p) \leq_{\mathcal{H}} recvSendToSucc_i(p,p')\ )$.
(A $recvSendToSucc$ operation on any peer p is initiated before it is completed.)

(j) $\forall p, p' \in \mathcal{P}\ (\ \forall i \in \mathcal{N}(sendToSucc_i(p,p') \in O_{\mathcal{H}}) \Rightarrow initSendToSucc_i(p) \in O_{\mathcal{H}} \wedge initSendToSucc_i(p) \leq_{\mathcal{H}} sendToSucc_i(p,p')\ )$.
(A $sendToSucc$ operation on any peer p is initiated before it is completed.)

(k) Semantics of `getSuccessor`*:*
$\forall p, p' \in \mathcal{P}\ (\ o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\ \exists o' \in O_{\mathcal{H}}\ initGetSucc_i(p) \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'\ )\ )$
(A $getSuccessor$ must return the correct *successor on the ring.)*

(l) Semantics of `sendToSuccessor`*:*
$\forall p, p' \in \mathcal{P}\ (\ o = recvSendToSucc_i(p',p) \in O_{\mathcal{H}} \Rightarrow \exists o' \in O_{\mathcal{H}}\ initSendToSucc_i(p) \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'\ )$
$\forall p, p' \in \mathcal{P}\ (\ o = sendToSucc_i(p',p) \in O_{\mathcal{H}} \Rightarrow (\ \exists o' \in O_{\mathcal{H}}\ initSendToSucc_i(p) \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'\ ) \wedge (\ \forall p'' \in \mathcal{P}, p'' \neq p' \nexists insert(p,p'') \in O_{\mathcal{H}} \wedge o' \leq_{\mathcal{H}} insert(p,p'') \leq_{\mathcal{H}} o\ )\ )$
(The message sent as part of $sendToSucc$ is always send to the correct *successor.)*

(m) Semantics of `INFOFROMPREDEVENT`*:*

i. $\forall p, p' \in \mathcal{P}\ (\forall i \in \mathcal{N}\ (infoFromPredEvent_i(p,p') \in O_{\mathcal{H}} \Rightarrow (initInfoFromPredEvent_i(p,p') \in O_{\mathcal{H}} \wedge initInfoFromPredEvent_i(p,p') \leq_{\mathcal{H}} infoFromPredEvent_i(p,p'))))$
*(*infoFromPredEvent *operation is initiated before is completed)*

ii. $\forall p, p' \in \mathcal{P}\ (\ \forall i \in \mathcal{N}\ (\ o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \neq \texttt{NULL} \wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\exists j \in \mathcal{N}\ (\ oinfo = infoFromPredEvent_j(p',p) \leq_{\mathcal{H}} o\ ) \wedge (\exists osucc \in O_{\mathcal{H}}\ (initGetSucc_i(p) \leq_{\mathcal{H}} osucc \leq_{\mathcal{H}} o \wedge oinfo \leq_{\mathcal{H}} osucc \wedge (\forall o' \in O_{\mathcal{H}}(oinfo \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} osucc \Rightarrow p' = succ_{\mathcal{H}_{o'}}(p)\ ))))\ ))$
(A $getSucc(p,p')$ operation with live peer p' returns only after $infoFromPredEvent(p',p)$ was processed at p', and p' is successor of p between $infoFromPredEvent$ and sometime before $getSucc$)

iii. $\forall p, p' \in \mathcal{P}\ (\ \forall i \in \mathcal{N}\ (\ o = sendToSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \neq \texttt{NULL} \wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\exists j \in \mathcal{N}\ (\ oinfo = infoFromPredEvent_j(p',p) \leq_{\mathcal{H}} o\ ) \wedge (\exists osucc \in O_{\mathcal{H}}\ (initSendToSucc_i(p) \leq_{\mathcal{H}} osucc \leq_{\mathcal{H}} o \wedge oinfo \leq_{\mathcal{H}} osucc \wedge (\forall o' \in O_{\mathcal{H}}(oinfo \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} osucc \Rightarrow p' = succ_{\mathcal{H}_{o'}}(p)\ ))))\ ))$
(A $sendToSucc(p,p')$ operation with live peer p' returns only after $infoFromPredEvent(p',p)$ was processed at p', and p' is successor of p between $infoFromPredEvent$ and sometime before $sendToSucc$)

(n) Semantics of `INFOFORSUCCEVENT`*:*

i. $\forall p \in \mathcal{P}(\forall i \in \mathcal{N}(infoForSuccEvent_i(p) \in O_{\mathcal{H}} \Rightarrow (initInfoForSuccEvent_i(p) \in O_{\mathcal{H}} \wedge initInfoForSuccEvent_i(p) \leq_{\mathcal{H}} infoForSuccEvent_i(p))))$
*(*infoForSuccEvent *operation is initiated before is completed)*

ii. $\forall p, p' \in \mathcal{P}\ (\ \forall i \in \mathcal{N}\ (\ o_1 = initInfoFromPredEvent_j(p',p) \in O_{\mathcal{H}} \Rightarrow (o_2 = infoForSuccEvent_j(p) \in O_{\mathcal{H}} \wedge infoForSuccEvent_j(p) \leq_{\mathcal{H}} initInfoFromPredEvent_j(p',p) \wedge \forall o \in O_{\mathcal{H}}(\ \neg(o_2 \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o1) \vee succ_{\mathcal{H}_o}(p) = p'\ ))\ )\ )$
(An $infoFromPredEvent_j(p',p)$ at peer p', the successor of p, is preceded by $infoForSuccEvent_j(p)$ at peer p.)

iii. $\forall p, p' \in \mathcal{P}, \forall i, j \in \mathcal{N}(o' = infoForSuccEvent_i(p) \wedge o'' =$

$infoForSuccEvent_j(p) \wedge succ_{\mathcal{H}_{o'}}(p) = p' \wedge succ_{\mathcal{H}_{o''}}(p) = p' \wedge o' <_{\mathcal{H}} o'' \Rightarrow \exists o \in O_{\mathcal{H}}(o' <_{\mathcal{H}} o <_{\mathcal{H}} o'' \wedge succ_{\mathcal{H}_o}(p) \neq p'))$
(Between two $infoForSuccEvent$ operations at peer p for the same successor p', the successor of p must have changed.)

(o) Semantics of `NEWSUCCEVENT`*:*

$\forall p, p' \in \mathcal{P}, \forall i \in \mathcal{N}((getSucc_i(p, p') \in O_{\mathcal{H}} \wedge p' \neq \texttt{NULL}) \Rightarrow (\exists j \in \mathcal{N} newSuccEvent_j(p, p') \leq_{\mathcal{H}} getSucc_i(p, p')))$

$\forall p, p' \in \mathcal{P}, \forall i \in \mathcal{N}(sendToSucc_i(p, p') \in O_{\mathcal{H}} \Rightarrow (\exists j \in \mathcal{N} newSuccEvent_j(p, p') \leq_{\mathcal{H}} sendToSucc_i(p, p')))$

($getSucc(p, p')$ or $sendToSucc(p, p')$ at peer p involve the successor p' only after $newSuccEvent(p, p')$ is thrown up.)

An *API ring history* (or simply an *API history*) $\mathcal{H}$ of operations on a set of peers $\mathcal{P}$ is a pair $(O_{\mathcal{H}}, \leq_{\mathcal{H}})$ with $O_{\mathcal{H}}$ a subset of operations on peers in $\mathcal{P}$ and $\leq_{\mathcal{H}}$ a partial order on these operations which satisfies the above mentioned properties. Note that $\mathcal{O}(\mathcal{P})$ is the set of operations under consideration.

The properties satisfied by an *API history* have been divided into two groups.

- *API restrictions*: These are the restrictions on how the API can be used. For example, restriction 3(b) says that all operations initiated at peer p in the history should be *after* $inserted(p)$.
- *Semantic restrictions* These are restrictions on operations which are not in the API users control. These are the properties that need to be satisfied by any implementation supporting the ring API in consideration.

Note that, given an *API ring history* $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, $\Pi_{O(\mathcal{P})}(\mathcal{H})$ is an *abstract ring history*.

Algorithm 8 : $p.initRing()$

```
⌈ 0: writeLock predPeerID, succList, state
| 1: // Set predPeerID, succList, state
| 2: predPeerID = p
| 3: for i = 0 to d − 1 do
| 4:     succlist[i] = SuccListEntry(p, JOINED, NOTSTAB);
| 5: end for
PinitRing(p)
| 6: state = JOINED
| 7: register JOINED msg handlers, JOINED stabilize
| 8: releaseLock state, predPeerID
⌊ 9: releaseLock succList
⌈ 10: // raise an event to signal end of insert
insert(p, p)
⌊ 11: raiseEvent(INSERT)
⌈ 12: // raise an event to signal end of insert
inserted(p) = inserted_0(p)
⌊ 13: raiseEvent(INSERTED)
```

Algorithm 9 : $p.initInsert(p')$

```
⌈ 0: writeLock state, succList
PinitInsert1(p, p')
| 1: if state ≠ JOINED then
⌊ 2:
⌈ 3:     releaseLock state, succList // Try again later
 PinitInsertAbort(p, p')
⌊ 4:     return;
  5: else
⌈ 6:     // Set state to INSERTING
| 7:     state = INSERTING
| 8:     releaseLock state
| 9: end if
| 10: // Insert p' into list as a JOINING peer
PinitInsert2(p, p')
| 11: succList.push_front(SuccListEntry(p', JOINING, NOTSTAB))
⌊ 12: releaseLock succList
```

Algorithm 10 : `RingJoinAckMsgHandler`

```
⌈ 0: get ackedPeerID from received message
| 1: readLock myPeerID
PcheckJack_i(p, p')
| 2: if !ackedPeerID.equals(myPeerID) then
| 3:     releaseLock myPeerID
| 4:     return;
| 5: end if
⌊ 6: releaseLock myPeerID
⌈ 7: writeLock succList, state
Pinsert1_i(p, p')
| 8: if state ≠ INSERTING then
⌊ 9:
⌈ 10:     releaseLock succlist, state // Abort insert
PinsertAbort_i(p, p')
⌊ 11:     return;
  12: else
⌈ 13:     state = JOINED
| 14: end if
Pinsert2_i(p, p')
| 15: succList[0].setState(JOINED);
| 16: succList[i].stabilized = NOTSTAB
        ∀0 ≤ i < succList.length
⌊ 17: succList.pop_back() // Remove last entry
⌈ 18: releaseLock state
Pinsert3_i(p, p')
⌊ 19: downgradeLock succList
⌈ 20: // Get data to be sent to p' from higher layers
initInsertEvent_i(p, p')
⌊ 21: raiseEvent(INSERT, p')
⌈ 22: //wait for return from event
insert_i(p, p')
⌊ 23: data = returned data;
⌈ 24: //send the data to p'
PsendJoin_i(p, p')
⌊ 25: ok = p'.RingJoinPeerMsgHandler(p, data)
⌈ 26: //release lock
Pinsert4_i(p, p')
⌊ 27: releaseLock succList
```

11.2 PEPPER Ring implementation

11.2.1 Operations We now present our implementation of the *ring* and identify the different operations in our implementation. Our implementation of $p.initRing(p)$, $p.initInsert(p')$, $p.initLeave()$, $p.initGetSucc()$ and $p.initSendToSucc(msg)$ are given in Algorithms 8, 9, 12, 21 and 19 respectively. $initRing(p)$ is the operation used to denote the invocation of the API method, $p.initRing()$. We similarly have operations $initInsert_i(p, p')$, $initLeave(p)$, $initGetSucc_i(p)$, $initSendToSucc_i(p)$ corresponding to API methods $p.initInsert(p')$, $p.initLeave()$, $p.initGetSucc()$, $p.initSendToSucc(msg)$ respectively.

In addition to the API methods, we have periodic procedures `RingPingSuccessor` and `JoinedRingStabilization` as specified in Algorithms 14 and 16. We also have message handlers `RingJoinAckMsgHanlder`, `RingLeaveAckMsgHandler`, `RingJoinPeerMsgHandler`, `RingPingMsgHandler`, `RingStabilizationMsgHandler` and `SendToSuccessorMsgHandler`, the code for which is given in Algorithms 10, 13, 11, 15, 18 and 20 respectively.

The operations we define are listed as part of the pseudocode for the above algorithms. In addition, we have the fail operation, $fail(p)$ $\forall p \in \mathcal{P}$. Given a set of peers $\mathcal{P}$, the set of allowed operations in any *PEPPER Ring History* is denoted by $\mathcal{O}_{\mathcal{PH}}$. Note that we do not model the operations in a `FREE` peer (for example, operations in Ring Stabilization in a `FREE` state are not considered.)

Before we define *PEPPER Ring History* corresponding

Algorithm 11 : `RingJoinPeerMsgHandler`

```
 0: get newSuccList, data from received msg
⌈ 1: writeLock succList, state
| 2: // Set succList, state
Pjoin_i(p', p)
| 3: succList = newSuccList
| 4: state = JOINED
| 5: register JOINED message handlers, JOINED stabilize
| 6: releaseLock state, succList
⌈ 7: // raise event to signal end of insert
inserted_i(p')
| 8: raiseEvent(INSERTED, data)
⌊ 9: return true
```

Algorithm 12 : $p.initLeave()$

```
⌈ 0: // Set state to LEAVING
| 1: writeLock state
PinitLeave1(p)
| 2: if state ≠ JOINED then
⌊ 3:
⌈ 4:     releaseLock state
PinitLeaveAbort(p)
⌊ 5:     return; // Unsuccessful leave
  6: else
⌈ 7:     unregister JOINED stabilize
PinitLeave2(p)
| 8:     state = LEAVING
⌊ 9:     releaseLock state
 10: end if
```

to our implementation of the ring, we first define the notion of *conflicting* operations. To do this we need the following notation: Given an operation o, o holds a *read lock* on resource R iff $o.r(R)$. Similarly, o holds a *write lock* on resource R iff $o.w(R)$.

Definition 10 (Conflicting operations) *Given operations o_1 and o_2, o_1* conflicts with *o_2 iff* $\exists R\ (o_1.w(R) \wedge o_2.w(R)) \vee (o_1.w(R) \wedge o_2.r(R)) \vee (o_1.r(R) \wedge o_2.w(R))$.

Two operations are conflicting iff they hold conflicting locks i.e one of the operations holds a *write lock* on some resource R and the other operations holds a *read lock* or a *write lock* on the same resource R.

Note that, given a history $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, $o_1, o_2 \in O_{\mathcal{H}} \wedge o_1$ *conflicts with* $o_2 \Rightarrow o_1 \leq_{\mathcal{H}} o_2 \vee o_2 \leq_{\mathcal{H}} o_1$.

Notation ($\mathcal{O}_{\mathcal{PH}}(p)$): $\mathcal{O}_{\mathcal{PH}}(p)$ is the subset of operations in $\mathcal{O}_{\mathcal{PH}}$ that occur on peer p.

Algorithm 13 : `RingLeaveAckMsgHandler`

```
⌈ 0: get ackedPeerID from received message
| 1: readLock myPeerID
PcheckLack_i(p, p')
| 2: if !ackedPeerID.equals(myPeerID) then
| 3:     releaseLock myPeerID
| 4:     return;
| 5: end if
⌊ 6: releaseLock myPeerID
⌈ 7: // Set state to FREE
| 8: readLock state
Pleft(p)
| 9: if state == LEAVING then
| 10:     unregister JOINED message handlers
| 11:     writeLock state
| 12:     state = FREE
| 13:     releaseLock state
| 14: else
| 15:     releaseLock state
| 16:     return;
| 17: end if
⌈ 18: // raise event to signal end of leave
leave(p)
⌊ 19: raiseEvent(LEAVE)
```

Algorithm 14 : `RingPingSuccessor`

0: // Infinite loop
1: **for** $i = 0$ to ∞ **do**
2: **repeat**
3: readLock $succList$
$Pping1_i(p, p')$
4: p' = copy of first `JOINED` successor
5: pos = position of p' is $p.succList$
6: releaseLock $succList$
7: p'.`RingPingMsgHandler`()
$Pping2_i(p, p')$
8:
9: writeLock $succList$
10: **if** no response & first `JOINED` successor = p' **then**
11: remove successor p' from $succList$;
12: // remove all `JOINING` entries that follow p'
13: **while** $p.succList[pos].state$ = `JOINING` **do**
14: $p_1 = p.succList[pos].peerid$
15: p_1.`RingPingMsgHandler`()
16: **if** no response **then**
17: remove pos pointer from $succList$
18: **else**
19: $p.succList[pos].state$ = `JOINED`
20: **end if**
21: **end while**
22: **end if**
$Pping3_i(p, p')$
23: releaseLock $succList$
24: **until** found live successor or $succList$ is empty;
25: // Ping first entry in succList if `LEAVING`
26: readLock $succList$
27: **if** $succList[0].state ==$ `LEAVING` **then**
28: get copy of first successor peerid (=p')
$Pping4_i(p, p')$
29: releaseLock $succList$
30: sendRecv PING msg to the `LEAVING` successor
$Pping5_i(p, p')$
31:
32: writeLock $succList$
33: **if** no response & 1^{st} succ is still the same **then**
34: remove successor;
35: **end if**
36: **end if**
$Pping6_i(p, p')$
37: releaseLock $succList$
38: **end for**

Algorithm 15 : `RingPingMsgHandler`

0: readLock $myPeerID$, $state$
1: Construct return message $retMsg$
$\forall i$ $Pping2_i(p, p')$, $Pping5_i(p, p')$
2: releaseLock $myPeerID$, $state$
3: return $retMsg$

Algorithm 16 : `Ring Stabilization - JOINED`

0: **for** $i = 0$ to ∞ **do**
1: // Update list based on successor's lists
2: readLock $state$, $succList$
$Pstab1_i(p, p')$
3: **if** $p.state ==$ `INSERTING` **then**
4: p' = first non-`LEAVING` peer in $p.succList$ after index 0
5: **else**
6: p' = first non-`LEAVING` peer in $p.succList$
7: **end if**
8: $indexp'$ = position of p' is $succList$
9: releaseLock $state$
10: //test if possibly new successor
$PstabTestNotStab_i(p, p')$
11: $data =$ `NULL`
12: **if** $p.succList[indexp'].stabilized ==$`NOTSTAB` **then**
13:
14: //get info for succ
$initInfoForSuccEvent_i(p)$
15: raiseEvent(`INFOFORSUCCEVENT`)
16: //wait for higher layers
$infoForSuccEvent_i(p)$
17: $data$ = returned data ($\neq$ `NULL`)
18: **end if**
19: //send stab msg
$Pstab2a_i(p, p')$
20: call p'.`RingStabilizationMsgHander`($p, data$)
21: //Wait for response
$Pstab2b_i(p, p')$
22: $rcvSuccList$ = response received from p'
23:
24: **if** p' does not respond **then**
25: releaseLock $succList$
$Pstab3_i(p, p')$
26: return
27: **end if**
28:
29: $updateSuccList(p', rcvSuccList)$
30: //check if need ack for joining
$Pstab5_i(p, p')$
31: p'=$succList[listLen - 2].peerid$
32: **if** $succList[listLen-2].state ==$`JOINING` **then**
33:
34: send join ack with `JOINING` peer p' to $p'' = succList[listLen - 3].peerid$
$Pstab6_i(p, p', p'')$
35:
36: // ELSE
$Pstab7_i(p, p')$
37: **else if** $succList[listLen - 2].state ==$`LEAVING` **then**
38:
39:
40: send a leave ack to `LEAVING` peer p'
$Pstab8_i(p, p', p'')$
41:
42: **end if**
43: releaseLock $succList$
$Pstab9_i(p, p')$
44:
45: **end for**

Algorithm 17 : $p.updateSuccList(p', rcvSuccList)$

upgradeWriteLock $succList$
Push `LEAVING` entries from $succList$ before p' in front of $rcvSuccList$
if $p.state ==$ `INSERTING` **then**
    Push $succList[0]$ in front of $rcvSuccList$
end if
// Remove last entry if list is too long
if number of `JOINED` entries in $rcvSuccList > d$ **then**
    Remove last `JOINED` entry.
    Delete other entries so that last entry is `JOINED`
end if
// Set succlist
$Pstab4_i(p, p')$
// Set $stabilized$ flags
set $stabilized$ for all $rcvSuccList$ to `NOTSTAB`
set $stabilized$ for first p' in $rcvSuccList$ to `STAB`
$succList$ = $rcvSuccList$;
// Handle `JOINING` & `LEAVING` peers
listLen = succList.length
if $succList[listLen - 1].state ==$`JOINING` $\vee$ `LEAVING` **then**
    $succList.pop\_back()$
end if

//raise newSuccEvent if needed
$PnewSuccEvent1_i(p, p')$
p'' = copy of first (`JOINED` or `JOINING`) and `STAB` entry in $succList$
if $p'' \neq$ `NULL` $\wedge\ p'' \neq lastNewSucc$ **then**

    //raise event $newSuccEvent_i(p, p')$
    $lastNewSucc = p'$
    $raiseEvent$(`NEWSUCCEVENT`,p')
end if
downgradeLock $succList$

Algorithm 18 : `RingStabilizationMsgHandler`

get $recvPredPeerID = p'$, $data$ from received message
$PprocessStabTestInfo_i(p, p')$
// Test if info from pred received
if $data \neq$ `NULL` **then**

    //raise event $initInfoFromPredEvent_i(p, p')$
    $raiseEvent$(`INFOFROMPREDEVENT`,$data$)
    //wait for event to be processed $infoFromPredEvent_i(p, p')$

end if
writeLock $predPeerID$
// Set predecessor peerid to that of the contacting peer
$predPeerID = recvPredPeerID$
$PprocessStab2_i(p, p')$
releaseLock $predPeerID$
readLock $succList, myPeerID, state$
//construct return message $retMsg$
$myState$ = `JOINED`
if $state ==$ `LEAVING` **then**
    $myState = state$
end if
$retMsg = succList$
$myEntry$ = SuccListEntry($p, myState,$ `NOTSTAB`)
$retMsg.push\_front(newEntry)$
releaseLock $succList, myPeerID, state$
return $retMsg$

Algorithm 19 : $p.initSendToSucc_i(msg)$

0: readLock $succList$
$PinitSendToSucc1_i(p, p')$
1: $p' =$ NULL
2: **for** i=0 to $succList.length$ **do**
3: **if** ($succList[i].state$ = JOINED) **then**
4: **if** $succList[i].stabilized$ = STAB **then**
5: $p' = succList[i].peerid$;
6: **end if**
7: break;
8: **end if**
9: **end for**
10:
11: //check if valid succ
$PinitSendToSuccTest_i(p, p')$
12: **if** $p' \neq$ NULL **then**
13:
14: sendRecv msg to p'; get response $retMsg$
$PinitSendToSucc2_i(p, p')$
15:
16: //check if response received
$PsendToSuccTest_i(p, p')$
17: **if** $retMsg$ received **then**
18:
19: // Throw up message to the higher layer
$sendToSucc_i(p, p')$
20: $raiseEvent$(SendToSucc, $retMsg$)
21: **end if**
22: **end if**
23: releaseLock $succList$
$PinitSendToSucc3_i(p, p')$
24:

Algorithm 20 : RingSendToSuccessorMsgHandler

0: readLock $succList$
$PrecvSendToSucc1_i(p, p')$
1:
2: // Throw up message to the higher layer
$recvSendToSucc_i(p, p')$
3: $retMsg = raiseEvent$(RecvSendToSucc, msg)
4: releaseLock $succList$
$PrecvSendToSucc2_i(p, p')$
5: return $retMsg$

Algorithm 21 : $p.initGetSucc_i()$

0: readLock $succList$
$PinitGetSucc1_i(p, p')$
1: $p' =$ NULL
2: **for** i=0 to $succList.length$ **do**
3: **if** ($succList[i].state$ = JOINED) **then**
4: **if** $succList[i].stabilized$ = STAB **then**
5: $p' = succList[i].peerid$;
6: **end if**
7: break;
8: **end if**
9: **end for**
10: releaseLock $succList$
11: return p'
$getSucc_i(p, p')$
12:

11.2.2 Definition We now present the definition of *PEPPER Ring History* $\mathcal{PH}$.

Definition 11 (PEPPER Ring History) *Given set of peers* $\mathcal{P}$ *and a set of allowed operations* $\mathcal{O}_{\mathcal{PH}}$ *on these peers,* $\mathcal{PH} = (O_{\mathcal{PH}}, \leq_{\mathcal{PH}})$ *is a* PEPPER Ring History *iff*

1. $\mathcal{PH}$ *is a history*
2. $O_{\mathcal{PH}} \subseteq \mathcal{O}_{\mathcal{PH}}$
3. API restrictions
   - *(a)* $\exists p \in \mathcal{P}$ ($initRing(p) \in O_{\mathcal{H}} \wedge insert(p,p) \in O_{\mathcal{H}} \wedge (\forall p' \in \mathcal{P}\ initRing(p') \in O_{\mathcal{H}} \Rightarrow p = p')$).
     (There exists a unique peer p *which starts off the ring.)*
   - *(b)* $\forall p, p' \in \mathcal{P}, p \neq p'$, ($initInsert(p,p') \in O_{\mathcal{H}} \Rightarrow (initRing(p') \notin O_{\mathcal{H}} \wedge (\forall p'' \in \mathcal{P}\ initInsert(p'',p') \Rightarrow p'' = p\ ))\ )$.
     (Insert of peer p' *is tried at most once. This is not a necessary but a convenient API restriction.)*
   - *(c)* $\forall p \in \mathcal{P}$ ($\forall op(p) \in \{initLeave(p), initGetSucc_i(p), initSendToSucc_i(p), \forall i \in \mathcal{N}\}$ $op(p) \in O_{\mathcal{H}} \Rightarrow \exists j \in \mathcal{N}(inserted_j(p) \in O_{\mathcal{H}} \wedge inserted_j(p) \leq_{\mathcal{H}} op(p)\ ) \wedge (\forall p' \in \mathcal{P}, \forall op(p,p') \in \{initInsert(p,p')\}$ ($op(p,p') \in O_{\mathcal{H}} \Rightarrow \exists j \in \mathcal{N}(inserted_j(p) \in O_{\mathcal{H}} \wedge inserted_j(p) \leq_{\mathcal{H}} op(p,p'))))$
     (All operations on peer p *except* $initRing(p)$ *are initiated after* $inserted_j(p), \forall j \in \mathcal{W}$*)*
   - *(d)* $\forall p \in \mathcal{P}$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow$ ($op(p) \in \{initRing(p), initLeave(p), initGetSucc_i(p), initSendToSucc_i(p), \forall i \in \mathcal{N}\} \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{H}} fail(p)$) $\wedge$ ($\forall p' \in \mathcal{P}$ ($op(p,p') \in \{initInsert(p,p')\}, \forall i \in \mathcal{N}\} \wedge op(p,p') \in O_{\mathcal{H}} \Rightarrow op(p,p') \leq_{\mathcal{H}} fail(p)$)))
     (All operations on peer p *are initiated before* $fail(p)$*.)*
   - *(e)* $\forall p \in \mathcal{P}$ ($leave(p) \in O_{\mathcal{H}} \Rightarrow$ ($op(p) \in \{initRing(p), initLeave(p), initGetSucc_i(p), initSendToSucc_i(p), \forall i \in \mathcal{N}\} \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{H}} leave(p)$) $\wedge$ ($\forall p' \in \mathcal{P}$ ($op(p,p') \in \{initInsert(p,p')\} \wedge op(p,p') \in O_{\mathcal{H}} \Rightarrow op(p,p') \leq_{\mathcal{H}} leave(p)$)))
     (All operations on peer p *are initiated before* $leave(p)$*.)*
   - *(f)* $\forall p \in \mathcal{P}$ ($leave(p) \in O_{\mathcal{H}} \Rightarrow$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow fail(p) \leq_{\mathcal{H}} leave(p)$)).
     (API fail operation on peer p *cannot occur after* $leave(p)$*)*

Table 3: Table of operations at peer p modifying $p.succList$

$PinitRing(p)$
$PinitInsert2(p,p')$
$Pinsert2(p,p')$
$Pjoin_i(p,p')$
$Pping3_i(p,p')$
$Pping6_i(p,p')$
$Pstab4_i(p,p')$

4. Happened Before Constraints

   Table 1 lists the operations in PEPPER Ring History $\mathcal{PH}$ *and for each operation specifies the operations (if any) that should occur before this operation in* $\mathcal{PH}$*. This table is generated directly from the pseudo-code for the algorithms. For example, in Algorithm 8,* $PinitRing(p)$ *occurs only after* $p.initRing()$ *is initiated i.e after operation* $initRing(p)$*. This is captured in row 2 of table 1.*

   Happened before *constraints can be derived from the* happened before *relationships specified in table 1. For example,*

   - $\forall p, p' \in \mathcal{P}, i \in \mathcal{N}$, $recvSendToSucc_i(p',p) \in O_{\mathcal{PH}} \Rightarrow initSendToSucc_i(p) \in O_{\mathcal{PH}} \wedge initSendToSucc_i(p) \leq_{\mathcal{PH}} recvSendToSucc_i(p',p)$
5. Conflict Constraints

   Table 2 shows the locks held by operations on a peer p*. For example, operation* $recvSendToSucc_i(p,p')$ *holds a* read lock *on resource* $succList$ *at peer* p *and also peer* p'*. All pairs of conflicting operations can be inferred from the locks table (Table 2).*

   Note from table 1 that $Pjoin_i(p,p') < inserted(p)$*. This defines the peer* p' *at which a readLock on* $succList$ *is held as part of the operation* $inserted(p)$ *(see table 2).*

   We also have conflicts which span a duration (henceforth referred to as duration conflicts*) i.e if all operations* between *operations* o_1 *and* o_2 *hold on to a lock on resource* R *which conflicts with operation* o*, then operation* o *cannot occur between operations* o_1 *and* o_2*. For example,* $insert(p,p')$ *cannot occur between operations* $PinitSendToSucc1_i(p,p')$ *and* $recvSendToSucc_i(p',p)$*. These conflicts can be inferred from tables 1 and 2.*

Chord-style ring uses the notion of *successor list* for fault tolerance. We use the following notation to denote a *successor list*.

Notation ($p.succList_{\mathcal{PH}}$**)**: Given *PEPPER history* $\mathcal{PH}$, $p.succList_{\mathcal{PH}}$ is the successor list of peer p. $p.succList_{\mathcal{PH}}.length$ is the length (number of pointers) of $p.succList_{\mathcal{PH}}$, and $p.succList_{\mathcal{PH}}[i]$ ($0 \leq i < p.succList_{\mathcal{PH}}.length$) refers to the i'th pointer

Table 1: Table of operations at peer p, operation(s) that need to *happen before* the given op.

Op	Parent op
$PinitRing(p)$	$initRing(p)$
$insert(p,p)$	$PinitRing(p)$
$inserted(p)$	$insert(p,p)$
$newSuccEvent_0(p,p)$	$inserted(p)$
$PinitRing2(p)$	$newSuccEvent_0(p,p)$
$PinitInsert1(p,p')$	$initInsert(p,p')$
$PinitInsert2(p,p')$	$PinitInsert1(p,p')$
$PinitInsertAbort(p,p')$	$PinitInsert1(p,p')$
$PcheckJack_i(p,p')$	$Pstab6_j(p'',p',p)$
$Pinsert1_i(p,p')$	$PcheckJack_i(p,p')$
$Pinsert2_i(p,p')$	$Pinsert1_i(p,p')$
$PinsertAbort_i(p,p')$	$Pinsert1_i(p,p')$
$Pinsert3_i(p,p')$	$Pinsert2_i(p,p')$
$initInsertEvent_i(p,p')$	$Pinsert3_i(p,p')$
$insert_i(p,p')$	$initInsertEvent_i(p,p')$
$PsendJoin_i(p,p')$	$insert_i(p,p')$
$Pjoin_i(p',p)$	$PsendJoin_i(p,p')$
$inserted_i(p)$	$Pjoin_i(p,p')$
$newSuccEvent_i(p,p')$	$inserted_i(p')$
$Pinsert4_i(p,p')$	$PsendJoin_i(p,p')$
$PinitLeave1(p)$	$initLeave(p)$
$PinitLeaveAbort(p)$	$PinitLeave1(p)$
$PinitLeave2(p)$	$PinitLeave1(p)$
$PcheckLack_i(p,p')$	$Pstab8_i(p'',p)$
$Pleft(p)$	$PcheckLack_i(p,p')$
$Pping2_i(p,p')$	$Pping1_i(p,p')$
$Pping3_i(p,p')$	$Pping2_i(p,p')$
$Pping4_i(p,p')$	$Pping3_i(p,p')$
$Pping5_i(p,p')$	$Pping4_i(p,p')$
$Pping6_i(p,p')$	$Pping5_i(p,p')$
$PstabTestNotStab_i(p,p')$	$Pstab1_i(p,p')$
$initInfoForSuccEvent_i(p)$	$PstabTestNotStab_i(p,p')$
$infoForSuccEvent_i(p)$	$initInfoForSuccEvent_i(p)$
$Pstab2a_i(p,p')$	$PstabTestNotStab_i(p,p')$
$PprocessStabTestInfo_i(p,p')$	$Pstab2a_i(p',p)$
$initInfoFromPredEvent_i(p,p')$	$PprocessStabTestInfo_i(p,p')$
$infoFromPredEvent_i(p,p')$	$initInfoFromPredEvent_i(p,p')$
$PprocessStab2_i(p,p')$	$PprocessStabTestInfo_i(p,p')$
$Pstab3_i(p,p')$	$Pstab2b_i(p,p')$
$Pstab4_i(p,p')$	$Pstab3_i(p,p')$
$Pstab5_i(p,p')$	$Pstab4_i(p,p')$
$Pstab6_i(p,p',p'')$	$Pstab5_i(p,p')$
$Pstab7_i(p,p')$	$Pstab6_i(p,p',p'')$
$Pstab8_i(p,p',p'')$	$Pstab7_i(p,p')$
$Pstab9_i(p,p')$	$Pstab8_i(p,p',p'')$
$PinitSendToSucc1_i(p,p')$	$initSendToSucc_i(p)$
$PinitSendToSuccTest_i(p,p')$	$PinitSendToSucc1_i(p,p')$
$PinitSendToSucc2_i(p,p')$	$PinitSendToSuccTest_i(p,p')$
$PsendToSuccTest_i(p,p')$	$PinitSendToSucc2_i(p,p')$
$sendToSucc_i(p,p')$	$PrecvSendToSucc2_i(p',p)$
$PinitSendToSucc3_i(p,p')$	$PinitSendToSuccTest_i(p,p')$
$PrecvSendToSucc1_i(p,p')$	$PinitSendToSucc2_i(p',p)$
$recvSendToSucc_i(p,p')$	$PrecvSendToSucc1_i(p,p')$
$PrecvSendToSucc2_i(p,p')$	$recvSendToSucc_i(p,p')$
$PinitGetSucc1_i(p,p')$	$initGetSucc_i(p)$
$getSucc_i(p,p')$	$PinitGetSucc1_i(p,p')$
$fail(p)$	$o \in \mathcal{O}_{\mathcal{PH}}(p)$, $o \neq fail(p)$
$leave(p)$	$o \in \mathcal{O}_{\mathcal{PH}}(p)$, $o \neq fail(p) \vee leave(p)$

Table 2: Table of operations at peer p and the locks each operation holds.

	$myPeerID$	$predPeerID$	$state$	$succList$
$PinitRing(p)$		w	w	w
$newSuccEvent_i(p, p')$				w
$PinitRing2(p)$				r
$PinitInsert1(p, p')$			w	w
$PinitInsertAbort(p, p')$			w	w
$PinitInsert2(p, p')$			w	w
$PcheckJack_i(p, p')$	r			
$Pinsert1_i(p, p')$			w	w
$PinsertAbort_i(p, p')$			w	w
$Pinsert2_i(p, p')$			w	w
$Pinsert3_i(p, p')$				r
$Pinsert4_i(p, p')$				r
$PsendJoin_i(p, p')$				r
$insert(p, p')$				r
$afterinsert(p, p')$				r
$Pjoin_i(p, p')$		w	w	w, r(p')
$(Pjoin_i(p, p') <) inserted_i(p)$				r(p')
$PinitLeave1(p)$			w	
$PinitLeaveAbort(p)$			w	
$PcheckLack_i(p, p')$	r			
$Pleft(p)$			w	
$Pping1_i(p, p')$				r
$Pping2_i(p, p')$	r(p')		r(p')	
$Pping3_i(p, p')$				w
$Pping4_i(p, p')$				r
$Pping5_i(p, p')$	r(p')		r(p')	
$Pping6_i(p, p')$				w
$Pstab1_i(p, p')$				r
$Pstab2a_i(p, p')$				r
$PprocessStab2_i(p, p')$	r	w	r	r
$Pstab2b_i(p, p')$				r
$Pstab3_i(p, p')$				r
$Pstab4_i(p, p')$				w
$Pstab5_i(p, p')$				r
$Pstab6_i(p, p', p'')$				r
$Pstab7_i(p, p')$				r
$Pstab8_i(p, p', p'')$				r
$Pstab9_i(p, p')$				r
$PinitSendToSucc1_i(p, p')$				r
$PinitSendToSucc2_i(p, p')$				r
$sendToSucc_i(p, p')$				r
$PinitSendToSucc3_i(p, p')$				r
$PrecvSendToSucc1_i(p, p')$				r, r(p')
$recvSendToSucc_i(p, p')$				r, r(p')
$PrecvSendToSucc2_i(p, p')$				r, r(p')
$PinitGetSucc1_i(p, p')$				r

Table 4: Table of operations at peer p modifying $p.state$

$PinitRing(p)$
$PinitInsert2(p, p')$
$Pinsert2(p, p')$
$Pjoin_i(p, p')$
$PinitLeave2(p)$

in $succList$. For convenience, we assume $\forall p \in \mathcal{P}\ p.succList[-1].peerid = p \wedge p.succList[-1].state = p.state$. We also maintain a $stabilized$ field for each pointer in the successor list. The value of this field can be `STAB` if this peer already contacted the corresponding peer as part of stabilization protocol, and `NOTSTAB` otherwise.

11.2.3 Unique insert We show that $\forall p, p' \in \mathcal{P}$ ($\forall i, j$ ($Pinsert2_i(p,p') \in O_{\mathcal{H}} \wedge Pinsert2_j(p,p') \in O_{\mathcal{H}} \Rightarrow i = j$)) and hence $\forall p, p' \in \mathcal{P}$($\forall i, j$ ($insert_i(p,p') \in O_{\mathcal{H}} \wedge insert_j(p,p') \in O_{\mathcal{H}} \Rightarrow i = j$)).

Lemma 2 *Given a* PEPPER Ring History $\mathcal{PH}$, $\forall p, p' \in \mathcal{P}$ ($\forall i, j$ ($insert_i(p,p') \in O_{\mathcal{PH}} \wedge insert_j(p,p') \in O_{\mathcal{PH}} \Rightarrow i = j$)).

*(*There is a unique i such that $insert_i(p,p') \in O_{\mathcal{PH}}$*)*

Proof: <Proof of this lemma uses claim 4>.

(*Proof sketch*: Proof uses the fact that $insert_i(p,p') \in O_{\mathcal{PH}}$ implies $Pinsert2_i(p,p') \in O_{\mathcal{PH}}$ and then relies on the fact from Claim 4 that there is a unique i such that $Pinsert2_i(p,p') \in O_{\mathcal{PH}}$)

Suppose $insert_i(p,p') \in O_{\mathcal{PH}} \wedge insert_j(p,p') \in O_{\mathcal{PH}}$.

From table 1, $insert_i(p,p') \in O_{\mathcal{PH}} \Rightarrow Pinsert2_i(p,p') \leq_{\mathcal{PH}} insert_i(p,p')$. Similarly, $Pinsert2_j(p,p') \leq_{\mathcal{PH}} insert_j(p,p')$

Using claim 4, $i = j$.

■

Claim 4 $\forall p, p' \in \mathcal{P}$ ($\forall i, j$ ($Pinsert2_i(p,p') \in O_{\mathcal{PH}} \wedge Pinsert2_j(p,p') \in O_{\mathcal{PH}} \Rightarrow i = j$)).

(*There is a unique i such that $Pinsert2_i(p,p') \in O_{\mathcal{PH}}$*)

Proof: <Proof of this claim uses claim 5>.

(*Proof sketch*: Proof by contradiction.)

Suppose $Pinsert2_i(p,p') \in O_{\mathcal{PH}} \wedge Pinsert2_j(p,p') \in O_{\mathcal{PH}} \wedge i \neq j$.

From table 2, $Pinsert2_i(p,p')$ *conflicts with* $Pinsert2_j(p,p')$. Without loss of generality, let's assume $Pinsert2_i(p,p') \leq_{\mathcal{PH}} Pinsert2_j(p,p')$.

From table 1, $Pinsert2_i(p,p') \in O_{\mathcal{H}} \Rightarrow Pinsert1_i(p,p') \leq_{\mathcal{PH}} Pinsert2_i(p,p')$. Similarly, $Pinsert1_j(p,p') \leq_{\mathcal{PH}} Pinsert2_j(p,p')$

$Pinsert1_j(p,p')$ has a *duration conflict* with $Pinsert1_i(p,p')$ and $Pinsert2_i(p,p')$. $Pinsert2_i(p,p') \leq_{\mathcal{PH}} Pinsert2_j(p,p')$ and therefore $Pinsert2_i(p,p') \leq_{\mathcal{PH}} Pinsert1_j(p,p')$.

Let $o' = Pinsert1_j(p,p')$. Using Claim 5, $p.succList_{\mathcal{PH}_{o'}}[0].peerid \neq p' \vee p.state_{\mathcal{PH}_{o'}} \neq$ INSERTING.

From implementation of $Pinsert1_j(p,p')$, $p' = p.succList_{\mathcal{PH}_{o'}}[0].peerid$. Therefore, $p.state_{\mathcal{PH}_{o'}} \neq$ INSERTING.

Again, from implementation of $Pinsert1_j(p,p')$, insert is aborted and hence $Pinsert2_j(p,p') \notin O_{\mathcal{PH}}$. Contradiction.

Therefore $i = j$.

■

Claim 5 $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall o' \in O_{\mathcal{PH}}$ ($o = Pinsert2_i(p,p') \in O_{\mathcal{PH}} \wedge o \leq_{\mathcal{PH}} o' \Rightarrow p' \neq p.succList_{\mathcal{PH}_{o'}}[0].peerid \vee p.state_{\mathcal{PH}_{o'}} \neq$ INSERTING))).

Proof: <Proof of this claim uses claims 6, 7 and 8>.

By induction on the number of operations in PEPPER Ring History $\mathcal{PH}$.

Base Case: $\mathcal{PH} = \{initRing(p), PinitRing(p), insert(p,p)\}$ (This is the shortest allowed *PEPPER Ring History*). In this case, the claim is trivially true.

Induction Hypothesis: Let's assume that the claim holds for any *PEPPER Ring History* $\mathcal{PH}'$ such that $|O_{\mathcal{PH}'}| = k$, $k \geq 3$.

Induction Step: We show that the claim holds for any *PEPPER Ring History* $\mathcal{PH}$ such that $|O_{\mathcal{PH}}| = k + 1$.

Let $op \in O_{\mathcal{PH}}, op \neq insert(p,p)$ for any $p \in \mathcal{P}$, such that $\nexists o' \in O_{\mathcal{PH}}$ ($op <_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - op$. Using claim 6, $\mathcal{PH}' = \Pi_{O'}(\mathcal{PH})$ is also a *PEPPER Ring History*.

$\mathcal{PH}'$ is a *PEPPER Ring History* and $|O_{\mathcal{PH}'}| = k$. Moreover $\forall o' \in O_{\mathcal{PH}}(o' \neq op \Rightarrow o' \in O_{\mathcal{PH}'})$. Therefore, using the induction hypothesis, $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall o' \in O_{\mathcal{H}}$ ($o' \neq op \wedge o = Pinsert2_i(p,p') \in O_{\mathcal{PH}'} \wedge o \leq_{\mathcal{PH}'} o' \Rightarrow p' \neq p.succList_{\mathcal{PH}'_{o'}}[0].peerid \vee p.state_{\mathcal{PH}'_{o'}} \neq$ INSERTING))).

$\forall p \in \mathcal{P}$ ($\forall o' \neq op$ ($p.succList_{\mathcal{PH}_{o'}}[0] = p.succList_{\mathcal{PH}'_{o'}}[0] \wedge p.state_{\mathcal{PH}_{o'}} = p.state_{\mathcal{PH}'_{o'}}$. Therefore, in *PEPPER history* $\mathcal{PH}$, the claim holds for all $o' \in O_{\mathcal{PH}}$ except possibly for $o' = op$.

Suppose $\nexists o = Pinsert2_i(p,p') \in O_{\mathcal{PH}'}$. The claim trivially holds in this case.

Now let $o = Pinsert2_i(p,p') \in O_{\mathcal{PH}'}$. If op does not modify $p.succList$ and $p.state$, $p.succList_{\mathcal{PH}_{op}}[0] = p.succList_{\mathcal{PH}'}[0] \wedge p.state_{\mathcal{PH}_{op}} = p.state_{\mathcal{PH}'}$. Using the induction hypothesis, we are through in this case.

We now consider only operations that modify $p.succList$ and $p.state$ (see tables 3, 4).

- $op = PinitRing(p)$.

  From table 1 $PinitRing(p) \in O_{\mathcal{PH}} \Rightarrow initRing(p) \leq_{\mathcal{PH}} PinitRing(p)$.

  Using API restriction 3(a), $insert(p,p) \in O_{\mathcal{PH}}$.

  From table 1, $insert(p,p) \in O_{\mathcal{PH}} \Rightarrow PinitRing(p) \leq_{\mathcal{PH}} insert(p,p)$.

  Therefore, $op < insert(p,p)$ contradicting the fact that $\nexists o' \in O_{\mathcal{PH}}$ ($op \leq_{\mathcal{PH}} o'$). So, $op \neq PinitRing(p)$.

- $op = PinitInsert2(p,p''')$:

  Using table 1 $\forall i \in \mathcal{N}$ ($o = Pinsert2_i(p,p') \in O_{\mathcal{PH}} \Rightarrow \exists j \in \mathcal{N}\ Pstab6_j(p'',p,p') \leq_{\mathcal{PH}} Pinsert2_i(p,p')$).

  Using claim 7, $Pstab6_j(p'',p,p') \in O_{\mathcal{PH}} \Rightarrow PinitInsert2(p,p') \leq_{\mathcal{PH}} Pstab6_j(p'',p,p')$.

  Using API restriction 3(b), $p''' \neq p'$.

  From implementation of $PinitInsert2(p,p''')$, $p.succList_{\mathcal{PH}_{op}}[0].peerid = p''' \neq p'$. We are therefore through in this case.

- $op = Pinsert2_j(p, p'')$:

  From Algorithm 10, $p.state_{\mathcal{PH}} = \texttt{JOINED} \neq \texttt{INSERTING}$. Noting that $\mathcal{PH} = \mathcal{PH}_o$, the claim holds in this case.

- $op = Pjoin_j(p, p'')$:

  From Algorithm 11, $p.state_{\mathcal{PH}} = \texttt{JOINED}$. The claim holds in this case.

- $Pping3_i(p, p'') \vee Pping6_i(p, p'')$:

  Note that $p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$.

  If $p.state_{\mathcal{PH}} \neq \texttt{INSERTING}$, we are through.

  Now suppose $p.state_{\mathcal{PH}} = \texttt{INSERTING} = p.state_{\mathcal{PH}'}$. Therefore, using the induction hypothesis, $p.succList_{\mathcal{PH}'}[0].peerid = p'$.

  From claim 8, $p.succList_{\mathcal{PH}}[0].state = \texttt{JOINING}$. Therefore, from Algorithm 14, $p'' \neq p.succList_{\mathcal{PH}}[0].peerid$. Therefore, $p.succList_{\mathcal{PH}}[0].peerid = p.succList_{\mathcal{PH}'}[0].peerid$.

  Hence $p.succList_{\mathcal{PH}}[0].peerid = p'$ and we are through in this case.

- $Pstab4_i(p, p'')$:

  $p.succList_{\mathcal{PH}}[0].peerid = p.succList_{\mathcal{PH}'}[0].peerid$ and $p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$. Therefore, using the induction hypothesis, we are through in this case.

- $op = PinitLeave2(p)$:

  From implementation of $PinitLeave2(p)$ (see Algorithm 12), $p.state_{\mathcal{PH}} = \texttt{LEAVING} \vee p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$ and $p.succList_{\mathcal{PH}}[0].peerid = p.succList_{\mathcal{PH}'}[0].peerid$.

  If $p.state_{\mathcal{PH}} = \texttt{LEAVING}$, then $p.state_{\mathcal{PH}} \neq \texttt{INSERTING}$ and hence the claim holds in this case.

  Otherwise, we are through by the induction hypothesis.

■

Claim 6 *Let $\mathcal{PH} = (O_{\mathcal{PH}}, \leq_{\mathcal{PH}})$ be a given* PEPPER Ring History. *Let $o \in O_{\mathcal{PH}}, o \neq insert(p)$ for some $p \in \mathcal{P}$, be an operation such that $\nexists o' \in O_{\mathcal{PH}}$ ($o \leq_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - o$. Then, $\Pi_{O'}(\mathcal{PH})$ is also a* PEPPER Ring History.

Proof: Given that $\mathcal{PH}$ is a *PEPPER Ring History*, it is still be a *PEPPER Ring History* if the operation o does not occur in the history. This is because no other operation in $O_{\mathcal{PH}}$ depends on o.

■

Claim 7 *Given* PEPPER Ring History $\mathcal{PH}$, $\forall p, p', p'' \in \mathcal{P}$ ($\forall j$ ($Pstab6_j(p'', p, p') \in O_{\mathcal{PH}} \Rightarrow PinitInsert2(p, p') \leq_{\mathcal{PH}} Pstab6_j(p'', p, p')$.

Proof: <Proof of this claim uses claim 9>.

From implementation of $Pstab6_j(p'', p, p')$, $o = Pstab6_j(p'', p, p') \in O_{\mathcal{PH}} \Rightarrow p' = p''.succList_{\mathcal{H}_o}[p''.succList_{\mathcal{PH}_o}.length - 2].peerid \wedge p'.state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}_o}[p''.succList_{\mathcal{H}_o}.length - 3].peerid$.

Using claim 9, $Pstab6_j(p'', p, p') \in O_{\mathcal{PH}} \Rightarrow PinitInsert2(p, p') \in O_{\mathcal{PH}}$. Clearly, $PinitInsert2(p, p') \leq_{\mathcal{PH}} Pstab6_j(p'', p, p')$.

■

Claim 8 *1.* $\forall p \in \mathcal{P}$ ($p.state_{\mathcal{PH}} = \texttt{INSERTING} \iff p.succList_{\mathcal{PH}}[0].state = \texttt{JOINING}$)

2. $\forall p \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($0 < i < p.succList_{\mathcal{PH}}.length \wedge p.succList_{\mathcal{PH}}[i-1].state \neq \texttt{JOINED} \Rightarrow p.succList_{\mathcal{PH}}[i].state \neq \texttt{JOINING}$))

Proof: By induction on the number of operations in PEPPER Ring History $\mathcal{PH}$.

Base Case: $\mathcal{PH} = \{initRing(p), PinitRing(p), insert(p, p)\}$ (This is the shortest allowed *PEPPER Ring History*). In this case, the claim is trivially true.

Induction Hypothesis: Let's assume that the claim holds for any *PEPPER Ring History* $\mathcal{PH}'$ such that $|O_{\mathcal{PH}'}| = k$.

Induction Step: We show that the claim holds for any *PEPPER Ring History* $\mathcal{PH}$ such that $|O_{\mathcal{PH}}| = k + 1$.

Let $op \in O_{\mathcal{PH}}, op \neq insert(p, p)$ for any $p \in \mathcal{P}$, such that $\nexists o' \in O_{\mathcal{PH}}$ ($op <_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - op$. Using claim 6, $\mathcal{PH}' = \Pi_{O'}(\mathcal{PH})$ is also a *PEPPER Ring History*.

$\mathcal{PH}'$ is a *PEPPER Ring History* and $|O_{\mathcal{PH}'}| = k$. Moreover $o' \in O_{\mathcal{PH}} \wedge o' \neq op \Rightarrow o' \in O_{\mathcal{PH}'}$. Therefore, using the induction hypothesis, $\forall p \in \mathcal{P}$ ($p.state_{\mathcal{PH}'} = \texttt{INSERTING} \Rightarrow p.succList_{\mathcal{PH}'}[0].state = \texttt{JOINING}$).

If op does not modify $p.succList$ and $p.state$, $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0] \wedge p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$. Using the induction hypothesis, we are through in this case.

We now consider only operations that modify $p.succList$ and $p.state$ (see tables 3, 4).

- $op = PinitInsert2(p, p''')$:

  From implementation of $PinitInsert2(p, p''')$, $p.state_{\mathcal{PH}} = \texttt{INSERTING} \wedge p.succList_{\mathcal{PH}}[0].state = \texttt{JOINING}$. We are therefore through with claim (1).

  $\forall i \in \mathcal{N}$ ($0 < i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p.succList_{\mathcal{PH}'}[i+1]$). Moreover, $p.succList_{\mathcal{PH}}[0].pid = p''' \wedge p.succList_{\mathcal{PH}}[0].state = \texttt{JOINING}$.

  For $i \neq 0$, claim (2) follows from induction hypothesis. We only need to show that $p.succList_{\mathcal{PH}}[1].state \neq \texttt{JOINING}$

  From table 1, $PinitInsert2(p, p''') \in O_{\mathcal{PH}} \Rightarrow o = PinitInsert1(p, p''') \leq_{\mathcal{PH}} PinitInsert2(p, p''')$.

From Algorithm 9, $p.state_{\mathcal{PH}_o}$ = JOINED. Using the induction hypothesis, $p.succList_{\mathcal{PH}_o}[0].state \neq$ JOINING.

Therefore $p.succList_{\mathcal{PH}}[1].state = p.succList_{\mathcal{PH}_o}[0].state \neq$ JOINING. Done with claim (2) in this case.

- $op = Pinsert2_j(p, p'')$:

  From Algorithm 10, $p.state_{\mathcal{PH}}$ = JOINED $\wedge$ $p.succList_{\mathcal{PH}}[0].state$ = JOINED. Claim (1) therefore holds in this case.

  $\forall i \in \mathcal{N}$ ($0 < i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i].state = p.succList_{\mathcal{PH}'}[i].state$). Moreover, $p.succList_{\mathcal{PH}}[0].state$ = JOINED. Using the induction hypothesis, we are through with claim (2) in this case.

- $op = Pjoin_j(p, p'')$:

  From table 1, $Pjoin_j(p, p'') \in O_{\mathcal{PH}} \Rightarrow o = Pinsert1_j(p'', p) \leq_{\mathcal{PH}} Pjoin_j(p, p'')$.

  From Algorithm 10, $p''.state_{\mathcal{PH}_o}$ = INSERTING.

  Using induction hypothesis, $p''.succList_{\mathcal{PH}_o}[0].state$ = JOINING $\wedge$ $p''.succList_{\mathcal{PH}_o}[1].state \neq$ JOINING.

  $\forall i \in \mathcal{N}$ ($0 \leq i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p''.succList_{\mathcal{PH}'}[i+1]$).

  By induction hypothesis, we are through with part (2) of the claim.

  $p.state_{\mathcal{PH}}$ = JOINED. To prove part (1) of the claim, we need to show that $p.succList_{\mathcal{PH}}[0].state \neq$ JOINING. From Algorithms 10 and 11, $p.succList_{\mathcal{PH}}[0] = p''.succList_{\mathcal{PH}_o}[1]$. Since, $p''.succList_{\mathcal{PH}_o}[1].state \neq$ JOINING (proved above, from the induction hypothesis), we are through with part (1).

- $op = Pping3_i(p, p'')$:

  Note that $p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$.

  Assume $p.state_{\mathcal{PH}} \neq$ INSERTING. Using the induction hypothesis, $p.succList_{\mathcal{PH}'}[0].state \neq$ JOINING.

  If $p.succList_{\mathcal{PH}'}[0].peerid \neq p''$, then $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$ and hence we are through with claim (1).

  If $p.succList_{\mathcal{PH}'}[0].peerid = p''$, from lines $10-21$ of algorithm 14, $p.succList_{\mathcal{PH}}[0].state \neq$ JOINING. We are therefore through with claim (1) in this case.

  Now suppose $p.state_{\mathcal{PH}}$ = INSERTING = $p.state_{\mathcal{PH}'}$. Using the induction hypothesis, $p.succList_{\mathcal{PH}'}[0].state$ = JOINING. Therefore, from Algorithm 14, $p'' \neq p.succList_{\mathcal{PH}}[0].peerid$. Therefore, $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$.

  Hence $p.succList_{\mathcal{PH}}[0].state$ = JOINING and we are through with claim (1) in this case.

  From lines $10-21$ of algorithm 14, we conclude claim (2) using the induction hypothesis.

- $op = Pping6_i(p, p'')$:

  Note that $p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$.

  Assume $p.state_{\mathcal{PH}} \neq$ INSERTING. Using the induction hypothesis, $p.succList_{\mathcal{PH}'}[0].state \neq$ JOINING.

  If $p.succList_{\mathcal{PH}'}[0].peerid \neq p''$, then $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$ and hence we are through with claim (1).

  Now assume $p.succList_{\mathcal{PH}'}[0].peerid = p''$. Since $p.succList_{\mathcal{PH}'}[0].state$ = LEAVING, using the induction hypothesis, $p.succList_{\mathcal{PH}}[0].state = p.succList_{\mathcal{PH}}[1].state \neq$ JOINING. We are therefore through with claim (1) in this case.

  Now suppose $p.state_{\mathcal{PH}}$ = INSERTING = $p.state_{\mathcal{PH}'}$. Using the induction hypothesis, $p.succList_{\mathcal{PH}'}[0].state$ = JOINING. Therefore, from Algorithm 14, $p'' \neq p.succList_{\mathcal{PH}}[0].peerid$. Therefore, $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$.

  Hence $p.succList_{\mathcal{PH}}[0].state$ = JOINING and we are through with claim (1) in this case.

  $p.succList_{\mathcal{PH}'}[i].peerid = p' \Rightarrow p.succList_{\mathcal{PH}'}[i].state$ = LEAVING. Using the induction hypothesis, $p.succList_{\mathcal{PH}'}[i+1].state \neq$ JOINING.

  Therefore, we conclude claim (2) using the induction hypothesis.

- $op = Pstab4_i(p, p'')$:

  Note that $p.state_{\mathcal{PH}} = p.state_{\mathcal{PH}'}$.

  Assume $p.state_{\mathcal{PH}} \neq$ INSERTING. Using the induction hypothesis, $p.succList_{\mathcal{PH}'}[0].state \neq$ JOINING. From Algorithm 17, either:

  -$p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$ $\wedge$ $p.state_{\mathcal{PH}'}$ = INSERTING (lines 2-4), which contradicts the assumption that $p.state_{\mathcal{PH}} \neq$ INSERTING,

  -or $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$ $\wedge$ $p.succList_{\mathcal{PH}'}[0].state$ = LEAVING (line 3). which implies $p.succList_{\mathcal{PH}}[0].state \neq$ JOINING,

  -or $p.succList_{\mathcal{PH}}[0].peerid = p.succList_{\mathcal{PH}'}[0].peerid = p''$. Since p'' replied to the stabilization message, $p''.state_{\mathcal{PH}'} \neq$ JOINING. In this case, from Algorithm 17 and Algorithm 18, $p.succList_{\mathcal{PH}}[0].state = p''.state_{\mathcal{PH}'}$ or $p.succList_{\mathcal{PH}}[0].state$ = JOINED if $p''.state_{\mathcal{PH}'}$ = INSERTING, so $p.succList_{\mathcal{PH}}[0].state \neq$ JOINING. This proves claim (1) in this case.

  We prove claim (2) now.

  Because $p.state_{\mathcal{PH}} \neq$ INSERTING, we have that $\exists j \in \mathcal{N}$ $\forall i \in \mathcal{N}$ (($0 \leq j \leq i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p''.succList_{\mathcal{PH}'}[i-j-1]$) $\wedge$

$(0 \leq i < j \Rightarrow p.succList_{\mathcal{PH}}[i].state = \texttt{LEAVING}))$. (from Algorithms 17 and 18)

Using induction hypothesis, we are through except possibly when $(j > 0 \wedge i = j-1) \vee i = j$.

For the case $j > 0 \wedge i = j-1$, we have that $p.succList_{\mathcal{PH}}[j-1].state = \texttt{LEAVING} \wedge p.succList_{\mathcal{PH}}[j].peerid = p''$. Since p'' answered the stabilization message, $p''.state_{\mathcal{PH}'} \neq \texttt{JOINING}$, so $p.succList_{\mathcal{PH}}[j].state \neq \texttt{JOINING}$.

Now, let us look at $i = j$. If $p.succList_{\mathcal{PH}}[j].state = \texttt{JOINED}$, the claim trivially holds.

Now suppose $p.succList_{\mathcal{PH}}[j].state = \texttt{LEAVING}$. Using induction hypothesis, $p''.state_{\mathcal{PH}'} = \texttt{LEAVING} \Rightarrow p''.succList_{\mathcal{PH}'}[0].state \neq \texttt{JOINING}$, Therefore, $p.succList_{\mathcal{PH}}[j+1].state = p''.succList_{\mathcal{PH}'}[0].state \neq \texttt{JOINING}$.

Therefore, we are through with claim (2) when $p.state_{\mathcal{PH}} \neq \texttt{INSERTING}$.

Now suppose $p.state_{\mathcal{PH}} = \texttt{INSERTING} = p.state_{\mathcal{PH}'}$. From Algorithm 16 (lines 19-21), $p.succList_{\mathcal{PH}}[0] = p.succList_{\mathcal{PH}'}[0]$. Using induction hypothesis, we are through with claim (1) in this case.

Using an argument similar to the one above, and using the induction hypothesis, we conclude that claim (2) holds in this case.

- $op = PinitLeave2(p)$:

  From implementation of $PinitLeave2(p)$ (see Algorithm 12), $p.state_{\mathcal{PH}} = \texttt{LEAVING}$ and $p.succList_{\mathcal{PH}} = p.succList_{\mathcal{PH}'}$.

  From table 1, $PinitLeave2(p) \in O_{\mathcal{PH}} \Rightarrow o = PinitLeave1(p) \leq_{\mathcal{PH}} PinitLeave2(p)$.

  From implementation of $PinitLeave2(p)$, $p.state_{\mathcal{PH}_o} = \texttt{JOINED}$. Using the induction hypothesis, $p.succList_{\mathcal{PH}_o}[0].state \neq \texttt{JOINING}$. Since $p.succList_{\mathcal{PH}_o} = p.succList_{\mathcal{PH}}$, claim (1) holds in this case.

  Otherwise, we are through by the induction hypothesis.

■

Claim 9 *Given* PEPPER Ring History $\mathcal{PH}$, $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}}.length \wedge p' = p''.succList_{\mathcal{PH}}[i].peerid \wedge p''.succList_{\mathcal{PH}}[i].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}}[i-1].peerid \Rightarrow PinitInsert2(p,p') \in O_{\mathcal{PH}}$)).

Proof: By induction on the number of operations in PEPPER Ring History $\mathcal{PH}$.

Base Case: $\mathcal{PH} = \{initRing(p), PinitRing(p), insert(p,p)\}$ (This is the shortest allowed *PEPPER Ring History*). In this case, the claim is trivially true.

Induction Hypothesis: Let's assume that the claim holds for any *PEPPER Ring History* $\mathcal{PH}'$ such that $|O_{\mathcal{PH}'}| = k$.

Induction Step: We show that the claim holds for any *PEPPER Ring History* $\mathcal{PH}$ such that $|O_{\mathcal{PH}}| = k+1$.

Let $op \in O_{\mathcal{PH}}$, such that $\nexists o' \in O_{\mathcal{PH}}$ ($op <_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - op$. Using claim 6, $\mathcal{PH}' = \Pi_{O'}(\mathcal{PH})$ is also a *PEPPER Ring History*.

$\mathcal{PH}'$ is a *PEPPER Ring History* and $|O_{\mathcal{PH}'}| = k$. Moreover $o' \in O_{\mathcal{PH}} \wedge o' \neq op \Rightarrow o' \in O_{\mathcal{PH}'}$. Therefore, using the induction hypothesis, $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}'}.length \wedge p' = p''.succList_{\mathcal{PH}'}[i].peerid \wedge p''.succList_{\mathcal{PH}'}[i].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}'}[i-1].peerid \Rightarrow PinitInsert2(p,p') \in O_{\mathcal{PH}'}$)).

If op does not modify $p.succList$, $p.succList_{\mathcal{PH}} = p.succList_{\mathcal{PH}'}$. Using the induction hypothesis, we are through in this case.

We now consider only operations that modify $p.succList$ (see table 3).

- $op = PinitInsert2(p,p')$:

  $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}'}.length \wedge p'' \neq p \Rightarrow p''.succList_{\mathcal{PH}'} = p''.succList_{\mathcal{PH}}$)). Moreover, from the implementation of $PinitInsert2(p,p')$, $\forall i$ ($0 < i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p.succList_{\mathcal{PH}'}[i-1]$).

  Therefore, using the induction hypothesis, we are through in this case.

  We only need to consider the case when $p'' = p \wedge i = 0$. From the implementation of $PinitInsert2(p,p')$, $p' = p.succList_{\mathcal{PH}}[0].peerid \wedge p.succList_{\mathcal{PH}}[0].state = \texttt{JOINING} \wedge p = p.succList_{\mathcal{PH}}[-1].peerid$. Since $PinitInsert2(p,p') \in O_{\mathcal{PH}}$, we are through in this case as well.

- $op = Pinsert2_j(p,p')$:

  $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}}.length \wedge p' = p''.succList_{\mathcal{PH}}[i].peerid \wedge p''.succList_{\mathcal{PH}}[i].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}}[i-1].peerid \Rightarrow p' = p''.succList_{\mathcal{PH}'}[i].peerid \wedge p''.succList_{\mathcal{PH}'}[i].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}'}[i-1].peerid$))) (i.e no new succList entries with JOINING state are introduced in history $\mathcal{PH}$).

  Using the induction hypothesis, we are through in this case.

- $op = Pjoin_j(p,p')$:

  $\forall p'' \in \mathcal{P}$ ($p'' \neq p \Rightarrow p''.succList_{\mathcal{PH}} = p''.succList_{\mathcal{PH}'}$). Moreover $p.succList_{\mathcal{PH}} = p'.succList_{\mathcal{PH}'}$.

  Using induction hypothesis, we are therefore through in this case.

- $op = Pping3_i(p, p')$:

  From lines $10 - 21$ in Algorithm 14, $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}}.length \wedge p' = p''.succList_{\mathcal{PH}}[i].peerid \wedge p''.succList_{\mathcal{PH}}[i].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}}[i-1].peerid \Rightarrow \exists j$ ($0 \leq j < p''.succList_{\mathcal{PH}'}.length \wedge p' = p''.succList_{\mathcal{PH}'}[j].peerid \wedge p''.succList_{\mathcal{PH}'}[j].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}'}[j-1].peerid$))) (i.e no new pairs of succList entries with right entry in JOINING state are introduced in history $\mathcal{PH}$).

  Using the induction hypothesis, we are through in this case.

- $op = Pping6_i(p, p')$:

  Using claim 8, we conclude that $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}}.length \wedge p' = p''.succList_{\mathcal{PH}}[i].peerid \wedge p''.succList_{\mathcal{PH}}[i].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}}[i-1].peerid \Rightarrow \exists j$ ($0 \leq j < p''.succList_{\mathcal{PH}'}.length \wedge p' = p''.succList_{\mathcal{PH}'}[j].peerid \wedge p''.succList_{\mathcal{PH}'}[j].state = \texttt{JOINING} \wedge p = p''.succList_{\mathcal{PH}'}[j-1].peerid$))) (i.e no new pairs of succList entries with right entry in JOINING state are introduced in history $\mathcal{PH}$).

  Using the induction hypothesis, we are through in this case.

- $op = Pstab4_i(p, p')$:

  $\forall p'' \in \mathcal{P}$ ($p'' \neq p \Rightarrow p''.succList_{\mathcal{PH}} = p''.succList_{\mathcal{PH}'}$).

  Let us see how $p.succList$ is affected by $Pstab4_i(p, p')$.

  If $p.state_{\mathcal{PH}} = \texttt{INSERTING}$, then $p.state_{\mathcal{PH}'} = \texttt{INSERTING} \wedge p.succList_{\mathcal{PH}}[0].state = p.succList_{\mathcal{PH}'}[0].state = \texttt{JOINING}$. From induction hypothesis, $PinitInsert2(p, p') \in O_{\mathcal{PH}}$. The pointers at positions > 0 are treated similar with the case when $p.state_{\mathcal{PH}} \neq \texttt{INSERTING}$, which we consider next.

  If $p.state_{\mathcal{PH}} \neq \texttt{INSERTING}$, then $\exists j \in \mathcal{N}\ \forall i \in \mathcal{N}$ (($0 \leq j \leq i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p''.succList_{\mathcal{PH}'}[i-j-1]$) $\wedge$ ($0 \leq i < j \Rightarrow p.succList_{\mathcal{PH}}[i].state = \texttt{LEAVING}$)). Here again no new pairs of succList entries with right entry in JOINING state are introduced in history $\mathcal{PH}$.

  Using the induction hypothesis, we are through in this case.

■

We henceforth drop the subscript for $Pinsert2$ and $insert$ operations and also all other operations following $Pinsert2$ in Algorithm 10.

11.2.4 Main Result

Theorem 1 (PEPPER Ring History is an API history) $\Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{PH}) = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ *is an API ring history.*

Proof: <Proof of this theorem relies on lemmas 3, 4, 5, 6 and 7>.

Using the definition of $\Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{PH})$, $o \in O_{\mathcal{PH}} \wedge o \in \mathcal{O}(\mathcal{P})$ iff $o \in O_{\mathcal{H}}$ and $o_1 \leq_{\mathcal{H}} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} \wedge o_1 \leq_{\mathcal{PH}} o_2$. Hence,

$$o \in \mathcal{O}(\mathcal{P}) \Rightarrow (o \in O_{\mathcal{PH}} \iff o \in O_{\mathcal{H}}) \quad (1)$$

$$o_1 \leq_{\mathcal{H}} o_2 \iff o_1, o_2 \in O_{\mathcal{H}} \wedge o_1 \leq_{\mathcal{PH}} o_2 \quad (2)$$

From the definition of $\mathcal{PH}$ and $\mathcal{H} = \Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{PH})$, we can conclude the first three properties of an *API history* for $\mathcal{H}$. We need to show that the *semantic restrictions* hold for $\mathcal{H}$.

- Using Lemma 3, we have property 4(a) - $\mathcal{H}_a = \Pi_{O(\mathcal{P})}(\mathcal{H})$ is an *abstract ring history*.
- From table 1, $fail(p) \in O_{\mathcal{PH}} \Rightarrow \forall o \in \mathcal{O}_{\mathcal{PH}}(p), o \neq fail(p), fail(p) \leq_{\mathcal{PH}} o$. Therefore, using equations (3) and (4) we conclude 4(b) - Any operation other than fail or leave involving p *happened before* $fail(p)$.
- From table 1, $leave(p) \in O_{\mathcal{PH}} \Rightarrow \forall o \in \mathcal{O}_{\mathcal{PH}}(p), o \neq fail(p), o \neq leave(p), leave(p) \leq_{\mathcal{PH}} o$. Therefore, using equations (3) and (4), we conclude 4(c) - Any operation other than fail or leave involving p *happened before* $leave(p)$.
- Using table 1, $\forall p \in \mathcal{P}$ ($insert(p,p) \in O_{\mathcal{PH}} \Rightarrow PinitRing(p) \in O_{\mathcal{PH}} \wedge PinitRing(p) \leq_{\mathcal{PH}} insert(p,p)$). Using table 1 again, $\forall p \in \mathcal{P}$ ($PinitRing(p) \in O_{\mathcal{PH}} \Rightarrow initRing(p) \in O_{\mathcal{PH}} \wedge initRing(p) \leq_{\mathcal{PH}} PinitRing(p)$). Hence, using equations (3) and (4), we conclude 4(d) - $\forall p \in \mathcal{P}$ ($insert(p,p) \in O_{\mathcal{H}} \Rightarrow initRing(p) \in O_{\mathcal{H}} \wedge initRing(p) \leq_{\mathcal{H}} insert(p,p)$).
- Using lemma 4, and using equations (3) and (4), we conclude 4(e) - $\forall p, p' \in \mathcal{P}, p \neq p'$, ($insert(p,p') \in O_{\mathcal{H}} \Rightarrow initInsert(p,p') \in O_{\mathcal{H}} \wedge initInsert(p,p') \leq_{\mathcal{H}} insert(p,p')$).
- From table 1, $\forall p \in \mathcal{P}$ ($inserted(p) \in O_{\mathcal{PH}} \Rightarrow \exists p'' \in \mathcal{P}$ ($insert(p'',p) \in O_{\mathcal{PH}} \wedge insert(p'',p) \leq_{\mathcal{PH}} inserted(p)$) or $inserted(p) \in O_{\mathcal{PH}} \Rightarrow insert(p,p) \in O_{\mathcal{PH}} \wedge insert(p,p) \leq_{\mathcal{PH}} inserted(p)$). Now using the previous two properties and using equations (3) and (4), we can conclude 4(f) - $\forall p \in \mathcal{P}$ ($inserted(p) \in O_{\mathcal{H}} \Rightarrow (initRing(p) \in O_{\mathcal{H}}) \vee (\exists p' \in \mathcal{P}\ initInsert(p',p) \in O_{\mathcal{H}} \wedge initInsert(p',p) \leq_{\mathcal{H}} inserted(p))$).
- Properties 4(g), 4(h), 4(i), 4(j) follow from table 1 and using equations (3) and (4).
- Correctness of $getSuccessor$ follows from corollary 1.1 of lemma 7.
- Correctness of $sendToSuccessor$ follows from corollary 1.2 of lemma 7.
- We show that conditions for correctness of $infoFromPredEvent$ are satisfied.
  - From Table 1, $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($infoFromPredEvent_i(p,p') \in O_{\mathcal{H}} \Rightarrow (initInfoFromPredEvent_i(p,p') \in O_{\mathcal{H}} \wedge initInfoFromPredEvent_i(p,p') \leq_{\mathcal{H}} infoFromPredEvent_i(p,p'))$)). This is the first correctness condition for $infoFromPredEvent$.
  - We need to show that the second correctness condition for $infoFromPredEvent$ holds: $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\wedge\ p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\exists j \in \mathcal{N}$ ($oinfo = infoFromPredEvent_j(p',p) \leq_{\mathcal{H}} o$) $\wedge$ ($\exists osucc \in O_{\mathcal{H}}$ ($initGetSucc_i(p) \leq_{\mathcal{H}} osucc \leq_{\mathcal{H}} o \wedge oinfo \leq_{\mathcal{H}} osucc \wedge$ ($\forall o' \in O_{\mathcal{H}}(oinfo \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} osucc \Rightarrow p' = succ_{\mathcal{H}_{o'}}(p)$)))))))

    From Lemma 5, $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\wedge\ p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\exists j \in \mathcal{N}$ ($oinfo = infoFromPredEvent_j(p',p) \leq_{\mathcal{H}} o \wedge (\nexists k \in \mathcal{N} oinfo <_{\mathcal{H}} infoFromPredEvent_k(p',p) <_{\mathcal{H}} o)$)))) (if there are multiple $infoFromPredEvent(p',p) <_{\mathcal{H}} o$, we just consider the last one). (*)

    From Corollary 1.1, $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\wedge\ p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\exists osucc = PinitGetSucc1_i(p,p') \in O_{\mathcal{H}} \wedge initGetSucc_i(p) \leq_{\mathcal{H}} osucc \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{osucc}}(p) = p')$)) (**)

    We prove now that $oinfo \leq_{\mathcal{H}} osucc$, where $oinfo$ and $osucc$ are defined in (*) and respectively (**).

    From Algorithm 21, operation $osucc = PinitGetSucc_j(p,p')$, $stabilized$ flag for the pointer corresponding to p' in $p.succList$ is STAB. Because that flag is set to STAB only in $Pstab4$ operation, we have that $\exists k \in \mathcal{N}(Pstab4_k(p,p') <_{\mathcal{H}} osucc) \wedge \exists i \in \mathcal{N} p.succList_{\mathcal{H}_{Pstab4_k(p,p')}}[i].stabilized =$ STAB $\wedge\ p.succList_{\mathcal{H}_{Pstab_k(p,p')}}[i].peerid = p'$. The $stabilized$ flag must have been NOTSTAB at some previous point, so there must be a $j1$ such that $infoForSuccEvent_{j1}(p) <_{\mathcal{H}} < infoFromPredEvent_{j1}(p',p) <_{\mathcal{H}} Pstab4_{j1}(p,p') \leq_{\mathcal{H}} Pstab4_k(p,p') <_{\mathcal{H}} osucc <_{\mathcal{H}} o$. From the second part

of (*), $j1 = j$ (we cannot have $infoFromPredEvent_{j1}(p', p) >_{\mathcal{H}} oinfo$, and two $infoFromPredEvent$ are ordered operations). So, $oinfo <_{\mathcal{H}} osucc$.

We have to prove that $\forall o' \in O_{\mathcal{H}}, oinfo \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} osucc \Rightarrow succ_{\mathcal{H}_{o'}}(p) = p'$.

We show first that $succ_{\mathcal{H}_{oinfo}}(p) = p'$.

From Lemma 6 and Table 1, $oinfo = infoFromPredEvent_j(p', p) \in O_{\mathcal{H}} \Rightarrow (oinfosucc = infoForSuccEvent_j(p) \in O_{\mathcal{H}} \wedge infoForSuccEvent_j(p) \leq_{\mathcal{H}} oinfo \wedge succ_{\mathcal{H}_o}(p) = p', \forall oinfosucc \leq_{\mathcal{H}} o \leq_{\mathcal{H}} oinfo)$. So we have that $succ_{\mathcal{H}_{oinfo}}(p) = p'$.

From (**) we have that $succ_{osucc}(p) = p'$. We prove by contradiction that $\forall o' \in O_{\mathcal{H}}, oinfo <_{\mathcal{H}} o' <_{\mathcal{H}} osucc \Rightarrow succ_{\mathcal{H}_{o'}}(p) = p'$.

Assume that $\exists o' \in O_{\mathcal{H}}(oinfo <_{\mathcal{H}} o' <_{\mathcal{H}} osucc \wedge succ_{\mathcal{H}_{o'}}(p) = p'' \neq p')$. Since $succ_{oinfo}(p) = p'$, $succ_{osucc}(p) = p'$ and $p' \in \mathcal{P}_{\mathcal{H}_i}$, p'' must have joined the ring at some point between $oinfo$ and $osucc$, so $o'' = insert(p, p'') \in O_{\mathcal{H}} \wedge oinfo <_{\mathcal{H}} o'' <_{\mathcal{H}} osucc$. From Algorithm 10, any operation that modifies $p.succList$ has a duration conflict with $Pinsert2(p, p'')$, $insert(p, p'')$. So, since in $Pinsert2(p, p'')$, the $stabilized$ flag for all pointers in $p.succList$ is set to NOTSTAB, the $stabilized$ flags are NOTSTAB for all pointers in $p.succList_{\mathcal{H}_{o''}}$. Let $indexp'$ be the first index in $p.succList_{\mathcal{H}_{osucc}}$ such that $p.succList_{\mathcal{H}_{osucc}}[indexp'].peerid = p'$. Since $p.succList_{\mathcal{H}_{osucc}}[indexp'].stabilized =$ STAB, there must $\exists k \in \mathcal{N}(o'' <_{\mathcal{H}} infoFromPredEvent_k(p, p') <_{\mathcal{H}} osucc)$. But $oinfo <_{\mathcal{H}} o''$, which implies $\exists k \in \mathcal{N}(oinfo <_{\mathcal{H}} infoFromPredEvent_k(p, p') <_{\mathcal{H}} osucc <_{\mathcal{H}} o)$. This contradicts (*), so out assumption that $\exists o' \in O_{\mathcal{H}}(oinfo <_{\mathcal{H}} o' <_{\mathcal{H}} osucc \wedge succ_{\mathcal{H}_{o'}}(p) = p'' \neq p')$ must be false.

We proved that $\forall o' \in O_{\mathcal{H}}, oinfo \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} osucc \Rightarrow succ_{\mathcal{H}_{o'}}(p) = p'$. (***)

Putting together (*), (**) and (***) we have that $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o = getSucc_i(p, p') \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow (\exists j \in \mathcal{N}$ ($oinfo = infoFromPredEvent_j(p', p) \leq_{\mathcal{H}} o$) $\wedge$ ($\exists osucc \in O_{\mathcal{H}}$ ($initGetSucc_i(p) \leq_{\mathcal{H}} osucc \leq_{\mathcal{H}} o \wedge oinfo \leq_{\mathcal{H}} osucc \wedge (\forall o' \in O_{\mathcal{H}}(oinfo \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} osucc \Rightarrow p' = succ_{\mathcal{H}_{o'}}(p)$)))))).

– Using the same argument as for $getSucc$ we can prove the correctness condition of $infoFromPredEvent$ related to $sendToSucc$.

- Correctness of $infoForSuccEvent$

  From Table 1, $\forall p \in \mathcal{P}(\forall i \in \mathcal{N}(infoForSuccEvent_i(p) \in O_{\mathcal{H}} \Rightarrow (initInfoForSuccEvent_i(p) \in O_{\mathcal{H}} \wedge initInfoForSuccEvent_i(p) \leq_{\mathcal{H}} infoForSuccEvent_i(p))))$. This is the first correctness condition for $infoForSuccEvent$.

  The second correctness condition follows from Lemma 6.

  We show now that $\forall p, p' \in \mathcal{P}, \forall i, j \in \mathcal{N}(o' = infoForSuccEvent_i(p) \wedge o'' = infoForSuccEvent_j(p) \wedge succ_{\mathcal{PH}_{o'}}(p) = p' \wedge succ_{\mathcal{PH}_{o''}}(p) = p' \wedge o' <_{\mathcal{PH}} o'' \Rightarrow \exists o \in O_{\mathcal{PH}}(o' <_{\mathcal{PH}} o <_{\mathcal{PH}} o'' \wedge succ_{\mathcal{PH}_o}(p) \neq p'))$.

  (Between two $infoForSuccEvent$ operations at peer p for the same successor p', the successor of p must have changed.)

  Proof sketch: during ring stabilization, $infoForSuccEvent$ is raised only if $stabilized$ flag of the pointer in $succList$ corresponding to the successor has value NOTSTAB. During stabilization, that flag will be set to STAB. In our case, we have two $infoForSuccEvent$s, so the $stabilized$ flag for the successor pointer must have been set to NOTSTAB, after if was STAB. This can only happen in Algorithm 10, when a new successor is inserted. So, we have a new successor between two $infoForSuccEvent$ operations at a peer p for the same successor p'.

- Correctness of $newSuccEvent$

  From Algorithms 21 and 19, $\forall p, p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o \in \{getSucc_i(p, p'), sendToSucc_i(p, p')\} \wedge o \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\Rightarrow \exists 0 \leq i < p.succList_{\mathcal{H}_o}.length \wedge p.succList_{\mathcal{H}_o}[i].peerid = p' \wedge p.succList_{\mathcal{H}_o}[i].stabilized =$ STAB. The only operation that sets the $stabilized$ flag to STAB is $Pstab4$. So, $p' = p.succList_{\mathcal{H}_o}[i].peerid \wedge p.succList_{\mathcal{H}_o}[i].stabilized =$ STAB $\Rightarrow \exists k \in \mathcal{H}Pstab4_k(p, p') <_{\mathcal{H}} o$.

  Because o defined above has a duration conflict with $Pstab4(p, p')$ and $newSuccEvent(p, p')$, since $Pstab4_k(p, p') <_{\mathcal{H}} o$, we have that $PnewSuccEvent1_k(p, p') <_{\mathcal{H}} o$.

  If $p' \neq_{\mathcal{H}_{PnewSuccEvent1_k(p,p')}} lastNewSucc$, we have that $newSuccEvent_k(p, p') <_{\mathcal{H}} o$ and this completes the proof. Else, since $lastNewSucc$ is only set in $newSuccEvent$ operation, $\exists k1 \in \mathcal{N} : newSuccEvent_k1(p, p') <_{\mathcal{H}} PnewSuccEvent1_k(p, p') <_{\mathcal{H}} o$.

  We showed that $\forall p, p' \in \mathcal{P}, \forall i \in \mathcal{N}((o \in \{getSucc_i(p, p'), sendToSucc_i(p, p')\} \wedge o \in O_{\mathcal{H}} \wedge p' \neq$ NULL$) \Rightarrow (\exists j \in \mathcal{N} newSuccEvent_j(p, p') \leq_{\mathcal{H}} o))$

We can therefore conclude that $\Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{PH}) = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ is an API ring history.

■

Lemma 3 (PEPPER Ring History is a ring history) $\Pi_{O(\mathcal{P})}(\mathcal{PH}) = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ *is a* ring history.

Proof: <Proof of this lemma relies on lemma 4>

Using the definition of $\Pi_{O(\mathcal{P})}(\mathcal{PH})$, $o \in O_{\mathcal{PH}} \wedge o \in O(\mathcal{P})$ iff $o \in O_{\mathcal{H}}$ and $o_1 \leq_{\mathcal{H}} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} \wedge o_1 \leq_{\mathcal{PH}} o_2$. Hence,

$$o \in O_{\mathcal{H}} \iff o \in O(\mathcal{P}) \wedge o \in O_{\mathcal{PH}} \quad (3)$$

$$o_1 \leq_{\mathcal{H}} o_2 \iff o_1, o_2 \in O_{\mathcal{H}} \wedge o_1 \leq_{\mathcal{PH}} o_2 \quad (4)$$

We now show that all the nine conditions for a *ring history* hold for $(O_{\mathcal{H}}, \leq_{\mathcal{H}})$.

- $(O_{\mathcal{H}}, \leq_{\mathcal{H}})$ is a history, by definition.
- Using equation (3), $O_{\mathcal{H}} \subseteq O(\mathcal{P})$.
- Using equation (3), $insert(p,p) \in O_{\mathcal{PH}} \iff insert(p,p) \in O_{\mathcal{H}}$.

  Using constraint 3, from API restriction 3(a), $\exists p \in \mathcal{P}$ ($initRing(p) \in O_{\mathcal{PH}} \wedge insert(p,p) \in O_{\mathcal{PH}} \wedge (\forall p' \in \mathcal{P}\ initRing(p') \in O_{\mathcal{PH}} \Rightarrow p = p')$).

  Hence, $\exists p \in \mathcal{P}\ insert(p,p) \in O_{\mathcal{PH}}$. From table 1, $insert(p',p') \in O_{\mathcal{PH}} \Rightarrow initRing(p') \in O_{\mathcal{PH}}$. Therefore, $p' = p$.

  Therefore, we conclude $\exists p \in \mathcal{P}$ ($insert(p,p) \in O_{\mathcal{H}} \wedge (\forall p' \in \mathcal{P}\ insert(p',p') \in O_{\mathcal{H}} \Rightarrow p = p')$) (condition 3 in the definition of ring history).
- From lemma 4, $insert(p,p') \in O_{\mathcal{PH}} \Rightarrow initInsert(p,p') \in O_{\mathcal{PH}} \wedge initInsert(p,p') \leq_{\mathcal{PH}} insert(p,p')$.

  From API constraint 3(c), $initInsert(p,p') \in O_{\mathcal{PH}} \Rightarrow inserted(p) \in O_{\mathcal{PH}} \wedge inserted(p) \leq_{\mathcal{PH}} initInsert(p,p')$.

  From table 1, $inserted(p) \in O_{\mathcal{PH}} \Rightarrow \exists p'' \in \mathcal{P}$ ($insert(p'',p) \in O_{\mathcal{PH}} \wedge insert(p'',p) \leq_{\mathcal{PH}} inserted(p)$) or $inserted(p) \in O_{\mathcal{PH}} \Rightarrow insert(p,p) \in O_{\mathcal{PH}} \wedge insert(p,p) \leq_{\mathcal{PH}} inserted(p)$.

  Therefore, using equations (1) and (2), we conclude that $\forall p,p' \in \mathcal{P}$ ($insert(p,p') \in O_{\mathcal{H}} \Rightarrow \exists p''$ ($insert(p'',p) \in O_{\mathcal{H}} \wedge insert(p'',p) \leq_{\mathcal{H}} insert(p,p')$)) (condition 4 in the definition of ring history).
- Now suppose $insert(p,p'), insert(p'',p') \in O_{\mathcal{PH}}, p \neq p'$. Let $p'' = p'$. From table 1, $insert(p',p') \in O_{\mathcal{PH}} \Rightarrow initRing(p') \in O_{\mathcal{PH}}$. From API constraint 3(b), $\forall p,p' \in \mathcal{P}, p \neq p'$, ($initInsert(p,p') \in O_{\mathcal{H}} \Rightarrow initRing(p') \notin O_{\mathcal{H}}$). Contradiction. Therefore $p'' \neq p'$ i.e $insert(p',p') \notin O_{\mathcal{PH}}$.

  From lemma 4, $initInsert(p,p'), initInsert(p'',p') \in O_{\mathcal{PH}}$. From API constraint 3(b), $\forall p,p' \in \mathcal{P}, p \neq p'$, ($initInsert(p,p') \in O_{\mathcal{H}} \Rightarrow (\forall p'' \in \mathcal{P}\ initRing(p'',p') \Rightarrow p'' = p)$). Therefore $\forall p,p' \in \mathcal{P}, p \neq p'$, ($insert(p,p') \in O_{\mathcal{H}} \Rightarrow (\forall p'' \in \mathcal{P}\ insert(p'',p') \in O_{\mathcal{H}} \Rightarrow p = p'')$).

  Therefore, using equations (1) and (2), we conclude $\forall p,p' \in \mathcal{P}, p \neq p'$, ($insert(p,p') \in O_{\mathcal{H}} \Rightarrow (insert(p',p') \notin O_{\mathcal{H}}) \wedge (\forall p'' \in \mathcal{P}\ insert(p'',p') \in O_{\mathcal{H}} \Rightarrow p = p'')$).
- From the locks table (see table 2), $\forall p,p',p''$ $Pinsert1(p,p')$ has a *duration conflict* with $Pinsert1(p,p'')$ and $insert(p,p'')$. Therefore, $\forall p,p',p'' \in \mathcal{P}$ ($insert(p,p'), insert(p,p'') \in O_{\mathcal{H}} \Rightarrow (insert(p,p') \leq_{\mathcal{H}} insert(p,p'')) \vee (insert(p,p'') \leq_{\mathcal{H}} insert(p,p'))$) (condition 6 in the definition of ring history).
- Suppose if possible $\exists p \in \mathcal{P}\ leave(p) \in O_{\mathcal{PH}} \wedge fail(p) \in O_{\mathcal{PH}}$.

  From table 1, $leave(p) \leq_{\mathcal{PH}} fail(p)$. From API constraint 3(f), $fail(p) \leq_{\mathcal{PH}} leave(p)$. Contradiction.

  Therefore, using equation (1), we can conclude $\forall p \in \mathcal{P}$ ($fail(p) \notin O_{\mathcal{H}} \vee leave(p) \notin O_{\mathcal{H}}$) (condition 7 in the definition of ring history).
- From table 1, $fail(p) \in O_{\mathcal{PH}} \Rightarrow \forall o \in \mathcal{O}_{\mathcal{PH}}(p), o \neq fail(p), fail(p) \leq_{\mathcal{PH}} o$. In particular, $fail(p) \in O_{\mathcal{PH}} \Rightarrow \forall p' \in \mathcal{P}\ insert(p,p') \in O_{\mathcal{PH}} \Rightarrow insert(p,p') \leq_{\mathcal{PH}} fail(p)$.

  Moreover, $fail(p) \in O_{\mathcal{PH}} \Rightarrow inserted(p) \in O_{\mathcal{PH}} \wedge inserted(p) \leq_{\mathcal{PH}} fail(p)$. Using the table again, $inserted(p) \in O_{\mathcal{PH}} \Rightarrow \exists p'' \in \mathcal{P}$ ($insert(p'',p) \in O_{\mathcal{PH}} \wedge insert(p'',p) \leq_{\mathcal{PH}} inserted(p)$) or $inserted(p) \in O_{\mathcal{PH}} \Rightarrow insert(p,p) \in O_{\mathcal{PH}} \wedge insert(p,p) \leq_{\mathcal{PH}} inserted(p)$.

  Therefore, using equations (1) and (2), $\forall p \in \mathcal{P}$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow (\forall p' \in \mathcal{P}\ insert(p,p') \in O_{\mathcal{H}} \Rightarrow insert(p,p') \leq_{\mathcal{H}} fail(p)) \wedge \exists p'$ ($insert(p',p) \in O_{\mathcal{H}} \wedge insert(p',p) \leq_{\mathcal{H}} fail(p)$)) (condition 8 in the definition of ring history).
- Same reasoning as above, condition 9 in the definition of ring history follows.

Hence, by definition, $(O_{\mathcal{H}}, \leq_{\mathcal{H}})$ is a ring history.

■

Lemma 4 *Given* PEPPER Ring History $\mathcal{H}$, $\forall p,p' \in \mathcal{P}$ ($p \neq p' \wedge insert(p,p') \in O_{\mathcal{H}} \Rightarrow initInsert(p,p') \in O_{\mathcal{H}} \wedge initInsert(p,p') \leq_{\mathcal{H}} insert(p,p')$)

Proof: <Proof of this lemma uses claim 7>.

Suppose $insert(p,p') \in O_{\mathcal{H}}$. From table 1, $\exists p'' \in \mathcal{P}\ \exists j \in \mathcal{N}\ Pstab6_j(p'',p,p') \in O_{\mathcal{H}} \wedge Pstab6_j(p'',p,p') \leq_{\mathcal{H}} insert(p,p')$

Using claim 7, $Pstab6_j(p'',p,p') \in O_{\mathcal{H}} \Rightarrow PinitInsert2(p,p') \in O_{\mathcal{H}} \wedge PinitInsert2(p,p') \leq_{\mathcal{H}} Pstab6_j(p'',p,p')$.

From table 1, $PinitInsert2(p,p') \in O_{\mathcal{H}} \Rightarrow initInsert(p,p') \in O_{\mathcal{H}} \wedge initInsert(p,p') \leq_{\mathcal{H}} PinitInsert2(p,p')$.

Therefore, $insert(p,p') \in O_{\mathcal{H}} \Rightarrow initInsert(p,p') \in O_{\mathcal{H}} \wedge initInsert(p,p') \leq_{\mathcal{H}} insert(p,p')$.

■

Lemma 5 $\forall p,p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o \in \{getSucc_i(p,p'), sendToSucc_i(p,p')\} \wedge o \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\Rightarrow \exists j \in \mathcal{N}$ ($infoFromPredEvent_j(p',p) \leq_{\mathcal{H}} o$))

(p' is returned as successor of p only after $infoFromPredEvent(p',p)$ is processed at peer p')

Proof: From Algorithms 21 and 19, $\forall p,p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o \in \{getSucc_i(p,p'), sendToSucc_i(p,p')\} \wedge o \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\Rightarrow \exists 0 \leq i < p.succList_{\mathcal{H}_o}.length \wedge p.succList_{\mathcal{H}_o}[i].peerid = p' \wedge p.succList_{\mathcal{H}_o}[i].stabilized =$ STAB. (`getStabilizedSucc()` method either returns NULL or returns $p' \neq$ NULL and the *stabilized* flag in the pointer corresponding to p' in *succList* is STAB). The only operation that sets the *stabilized* flag to STAB is $Pstab4$. So, $p' = p.succList_{\mathcal{H}_o}[i].peerid \wedge p.succList_{\mathcal{H}_o}[i].stabilized =$ STAB $\Rightarrow \exists k \in \mathcal{H} Pstab4_k(p,p') <_{\mathcal{H}} o$. (*)

From Algorithm 16, $\exists j \in \mathcal{N} infoForSuccEvent_j(p) <_{\mathcal{H}} Pstab4_j(p,p') \leq_{\mathcal{H}} Pstab4_k(p,p')$ (because the pointer corresponding to p' must have had the *stabilized* flag = NOTSTAB before being set to STAB). (**)

From Table 1 and Algorithm 18, $Pstab4_j(p,p') \in O_{\mathcal{H}} \wedge infoForSuccEvent_j(p) \in O_{\mathcal{H}} \Rightarrow infoFromPredEvent_j(p',p) <_{\mathcal{H}} Pstab4_j(p,p')$. (***)

From (*), (**) and (***) we conclude that $\forall p,p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o \in \{getSucc_i(p,p'), sendToSucc_i(p,p')\} \wedge o \in O_{\mathcal{H}} \wedge p' \neq$ NULL $\Rightarrow \exists j \in \mathcal{N}$ ($infoFromPredEvent_j(p',p) \leq_{\mathcal{H}} o$))). This is the second correctness condition for *infoFromPredEvent*.

■

Lemma 6 $\forall p,p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o_1 = initInfoFromPredEvent_j(p',p) \in O_{\mathcal{H}} \Rightarrow (o_2 = infoForSuccEvent_j(p) \in O_{\mathcal{H}} \wedge infoForSuccEvent_j(p) \leq_{\mathcal{H}} initInfoFromPredEvent_j(p',p) \wedge \forall o \in O_{\mathcal{H}} (\neg(o_2 \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_1) \vee succ_{\mathcal{H}_o}(p) = p'))$))

Proof: From Table 1, and Algorithm 18, $initInfoFromPredEvent_i(p',p) \Rightarrow (Pstab2a_i(p,p') <_{\mathcal{H}} initInfoFromPredEvent_i(p',p) \wedge infoForSuccEvent_i(p) \in O_{\mathcal{H}})$. From Table 1 and Algorithm 16, $infoForSuccEvent_i(p) <_{\mathcal{H}} Pstab2a_i(p,p')$, so $initInfoFromPredEvent_i(p',p) \Rightarrow o = infoForSuccEvent_i(p) <_{\mathcal{H}} initInfoFromPredEvent_i(p,p')$.

From Claim 8, part 2, and Algorithm 16, lines 3-7, if $o_2 = infoForSuccEvent_i(p)$ and $indexp'$ is the position of p' in $p.succList_{\mathcal{H}_o}$, then $p.succList_{\mathcal{H}_{o_2}}[indexp'].state =$ JOINED. Since p' is the first pointer in $p.succList_{\mathcal{H}_{o_2}}$ that corresponds to a JOINED peer, and p' is live in $\mathcal{H}_{o_2}$, $p' = p.trimList_{\mathcal{H}_{o_2}}[0]$. From Lemma 7, $p' = succ_{\mathcal{H}_{o_2}}(p)$.

Between o_2 and o_1, no new successor could have been inserted, since $Pinsert2(p,p'')$ has a duration conflict with o_2, o_1. Peer p' is live at o_1, so $p' = succ_{\mathcal{H}_o}(p), \forall o \in O_{\mathcal{H}}, o_2 \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_1$.

We proved that $\forall p,p' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($o_1 = initInfoFromPredEvent_j(p',p) \in O_{\mathcal{H}} \Rightarrow (o_2 = infoForSuccEvent_j(p) \in O_{\mathcal{H}} \wedge infoForSuccEvent_j(p) \leq_{\mathcal{H}} initInfoFromPredEvent_j(p',p) \wedge \forall o \in O_{\mathcal{H}} s.to_2 \leq_{\mathcal{H}} o \leq_{\mathcal{H}} (o_1 succ_{\mathcal{H}_o}(p) = p'))$)).

■

11.2.5 Consistent Successor Pointers From previous lemma, $\Pi_{O(\mathcal{P})}(\mathcal{PH}) = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ is a *ring history* and hence we can talk about the induced ring $R_{\mathcal{PH}} = (\mathcal{P}_{\mathcal{H}}, succ_{\mathcal{H}} : \mathcal{P}_{\mathcal{H}} \rightarrow \mathcal{P}_{\mathcal{H}})$.

Definition 12 ($p.trimList_{\mathcal{H}}$) *Given a* PEPPER Ring History $\mathcal{PH}$*, we define* $p.trimList_{\mathcal{PH}}$ *as the* trimmed *copy of* $p.succList_{\mathcal{PH}}$ *with only peerids of entries corresponding to* live *peers in the ring* $R_{\mathcal{PH}}$.

Lemma 7 (Consistent Successor Pointers) *Given a* PEPPER Ring History $\mathcal{PH}$*,* $\Pi_{O(\mathcal{P})}(\mathcal{PH})$ *is a ring history. Moreover,* $\forall p \in \mathcal{P}_{\mathcal{PH}}$ ($\forall i$ ($0 \leq i < p.trimList.length - 1 \Rightarrow succ_{\mathcal{PH}}(p.trimList_{\mathcal{PH}}[i]) = p.trimList_{\mathcal{PH}}[i+1] \wedge succ_{\mathcal{PH}}(p) = p.trimList_{\mathcal{PH}}[0]$)).

Proof: <Proof of this lemma relies on lemmas 3 and 8>.

From Lemma 3, $\Pi_{O(\mathcal{P})}(\mathcal{PH})$ is a ring history. We prove by induction on the length of the PEPPER Ring History $\mathcal{PH}$ that $\forall p \in \mathcal{P}_{\mathcal{PH}}$ ($\forall i$ ($0 \leq i < p.trimList.length - 1 \Rightarrow succ_{\mathcal{PH}}(p.trimList_{\mathcal{PH}}[i]) = p.trimList_{\mathcal{PH}}[i+1] \wedge succ_{\mathcal{PH}}(p) = p.trimList_{\mathcal{PH}}[0]$)).

Base case: $O_{\mathcal{PH}} = \{initRing(p), PinitRing(p), insert(p)\}$, for some $p \in \mathcal{P}$. Then $\forall i, 0 \leq i < p.trimList.length - 1, p.trimList_{\mathcal{PH}}[i] = p.trimList_{\mathcal{PH}}[i+1] = p$ and $succ_{\mathcal{H}}(p) = p.trimList_{\mathcal{H}}[0] = p$. The result therefore holds in this case.

Induction Hypothesis: Assume that the result holds for any *PEPPER Ring History* $\mathcal{PH}'$ such that $|O_{\mathcal{PH}'}| = k$.

Induction Step: We now show using the induction hypothesis that the result holds for any *PEPPER Ring History* $\mathcal{PH}$ such that $|O_{\mathcal{PH}}| = k + 1$.

Let $o \in O_{\mathcal{PH}}$, such that $\nexists o' \in O_{\mathcal{PH}}$ ($o <_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - o$. From claim 6, $\mathcal{PH}' = \Pi_{O'}(\mathcal{PH})$ is also a *PEPPER Ring History*.

Since $\mathcal{PH}'$ is a *PEPPER Ring History* and $|O_{\mathcal{PH}'}| = k$, using the induction hypothesis, $\forall p \in \mathcal{P}_{\mathcal{PH}'}$ (($\forall i, 0 \leq i < p.trimList.length$, $succ_{\mathcal{PH}'}(p.trimList_{\mathcal{PH}'}[i]) = p.trimList_{\mathcal{PH}'}[i+1]$) $\wedge$ $succ_{\mathcal{PH}'}(p) = p.trimList_{\mathcal{PH}'}[0]$).

We now show that the invariant holds in *PEPPER Ring History* $\mathcal{PH}$ considering different possibilities for operation o. Without loss of generality, we consider operations $o(p)$ and $o(p, p')$ for some peers p, p'.

- $o \in \mathcal{O}_{\mathcal{PH}}$, $o \neq insert(p)$, $insert(p, p')$, $Pjoin(p, p')$, $leave(p)$, $fail(p)$, $Pstab4_i(p, p')$ $\forall i \in \mathcal{N}$.

  $succ_{\mathcal{PH}} = succ_{\mathcal{PH}'}$; $\mathcal{P}_{\mathcal{PH}} = \mathcal{P}_{\mathcal{PH}'}$; $\forall p'' \in P_{\mathcal{PH}}$ $p''.trimList_{\mathcal{PH}} = p''.trimList_{\mathcal{PH}'}$.

  Using the induction hypothesis, the invariant holds for *PEPPER history* $\mathcal{PH}$ in this case.

- $o = insert(p, p')$

  $\mathcal{P}_{\mathcal{PH}} = \mathcal{P}_{\mathcal{PH}'} \cup \{p'\}$. Moreover, $succ_{\mathcal{PH}}(p) = p', succ_{\mathcal{PH}}(p') = succ_{\mathcal{PH}'}(p) = p_s$ and $succ_{\mathcal{PH}}(p'') = succ_{\mathcal{PH}'}(p''), \forall p'' \in \mathcal{P}_{\mathcal{PH}}, p'' \neq p, p'$.

  We want to use lemma 8. Since $o = insert(p, p') \in O_{\mathcal{PH}}$, from Table 1, $o' = Pinsert2(p, p') \in O_{\mathcal{PH}} \wedge o' \leq_{\mathcal{PH}} o$. So, we can apply lemma 8 for operation o'. However, from the fact that there is a duration lock from $Pinsert2(p, p')$ to $insert(p, p')$, no operation can modify $succList$ between o' and o, so the lemma holds at o. $\forall p'' \in \mathcal{P}_{\mathcal{PH}}$ (($\forall i, 0 \leq i < p''.trimList.length, p''.trimList_{\mathcal{PH}}[i] = p' \Rightarrow p''.trimList_{\mathcal{PH}}[i-1] = p$).

  - $\forall p'' \in \mathcal{P}_{\mathcal{PH}}, p'' \neq p, p'$:

    Since $\mathcal{P}_{\mathcal{PH}} = \mathcal{P}_{\mathcal{PH}'} \cup \{p'\}$, only entries corresponding to p' are possible new entrants into $p''.trimList_{\mathcal{PH}}$ (when compared with $p''.trimList_{\mathcal{PH}'}$).

    If $\forall i, 0 \leq i < p''.trimList_{\mathcal{PH}}.length$ ($p''.trimList_{\mathcal{PH}}[i] \neq p'$), then $p''.triList_{\mathcal{PH}} = p''.trimList_{\mathcal{PH}'}$ and from induction hypothesis, the result holds.

    Now, we consider the case when $\exists j, 0 \leq j < p''.trimList_{\mathcal{PH}}.length$($p''.trimList_{\mathcal{PH}}[j] = p'$).

    Lets consider the pairs $i, i+1$ such that $p''.trimList_{\mathcal{PH}}[i], p''.trimList_{\mathcal{PH}}[i+1] \neq p'$. Using Lemma 8, $p''.trimList_{\mathcal{PH}}[i+1] \neq p' \Rightarrow p''.trimList_{\mathcal{PH}}[i] \neq p$.

    $p''.trimList_{\mathcal{PH}}[i] \neq p, p' \Rightarrow$ ($\exists j$ $p''.trimList_{\mathcal{PH}}[i] = p''.trimList_{\mathcal{PH}'}[j] \wedge p''.trimList_{\mathcal{PH}}[i+1] = p''.trimList_{\mathcal{PH}'}[j+1]$) (since only entries corresponding to p' are possible new entrants into $p''.trimList_{\mathcal{PH}}$).

    Using induction hypothesis, $succ_{\mathcal{PH}'}(p.trimList_{\mathcal{PH}'}[j]) = p.trimList_{\mathcal{PH}'}[j+1]$. Since $p.trimList_{\mathcal{PH}'}[j] = p.trimList_{\mathcal{PH}}[i] \neq p, p'$, $succ_{\mathcal{PH}'}(p.trimList_{\mathcal{PH}'}[j]) = succ_{\mathcal{PH}}(p.trimList_{\mathcal{PH}}[i])$. Hence, $succ_{\mathcal{PH}}(p.trimList_{\mathcal{PH}}[i]) = p.trimList_{\mathcal{PH}}[i+1]$).

    Now consider the pairs $i, i+1$ such that $p''.trimList_{\mathcal{PH}}[i] = p'$. From Lemma 8, $p''.trimList_{\mathcal{PH}}[i] = p' \Rightarrow p''.trimList_{\mathcal{PH}}[i-1] = p$.

    Using induction hypothesis, $succ_{\mathcal{PH}'}(p''.trimList_{\mathcal{PH}'}[i-1]) = p''.trimList_{\mathcal{PH}'}[i] = p_s$, for some p_s.

    From how $succ$ and $trimList$ are defined, since $p''.trimList_{\mathcal{PH}}[i] = p'$ (this entry was inserted in the $p''.trimList$ due to $insert(p, p')$ operation), we have that $p''.trimList_{\mathcal{PH}}[i+1] = p''.trimList_{\mathcal{PH}'}[i] = p_s \wedge p''.trimList_{\mathcal{PH}}[i-1] = p''.trimList_{\mathcal{PH}'}[i-1] = p$. Therefore, $p_s = succ_{\mathcal{PH}'}(p) = p''.trimList_{\mathcal{PH}}[i+1]$.

Since $succ_{\mathcal{PH}}(p) = p' \wedge succ_{\mathcal{PH}}(p') = p_s$, by the definition of $succ$, we are through in this case.

This proves first part of the claim.

$\forall p'' \in \mathcal{P}_{\mathcal{PH}}, p'' \neq p, p', p''.trimList_{\mathcal{PH}}[0] = p''.trimList_{\mathcal{PH}'}[0]$. Using the induction hypothesis, $p''.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p'')$. Since $succ_{\mathcal{PH}'}(p'') = succ_{\mathcal{PH}}(p'')$, we have the second part of the claim.

– $p'' = p'$:

From Algorithm 10, $p'.trimList_{\mathcal{PH}}$ is still uninitialized, and hence trivially satisfies the invariant.

– $p'' = p$:

$\forall i \quad 0 < i < p.trimList_{\mathcal{PH}}.length, p.trimList_{\mathcal{PH}}[i] = p.trimList_{\mathcal{PH}'}[i-1]$. $p.trimList_{\mathcal{PH}}[0] = p'$. Using the same argument as in the first subcase, we can prove the first part of the claim ($\forall i, 0 < i < p.trimList_{\mathcal{PH}}.length$).

For $i = 0$, $p.trimList_{\mathcal{PH}}[0] = p'$ and $p.trimList_{\mathcal{PH}}[1] = p.trimList_{\mathcal{PH}'}[0]$. Using the induction hypothesis, $p.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p) = p_s$. Since $succ_{\mathcal{PH}}(p') = p_s$, we are through with the first part of the claim.

$p.trimList_{\mathcal{PH}}[0] = p' = succ_{\mathcal{PH}}(p)$. This proves the second part of the claim.

- $o = Pjoin(p', p)$

In this case, $succ_{\mathcal{PH}} = succ_{\mathcal{PH}'}$; $\mathcal{P}_{\mathcal{PH}} = \mathcal{P}_{\mathcal{PH}'}$; $\forall p'' \in P_{\mathcal{PH}}\ p'' \neq p'\ p''.trimList_{\mathcal{PH}} = p''.trimList_{\mathcal{PH}'}$.

Using the induction hypothesis, the invariant holds for $\forall p'' \in P_{\mathcal{PH}}\ p'' \neq p'$.

$p'.trimList_{\mathcal{PH}}$ is same as $p.trimList_{\mathcal{PH}'}$ with possible new entries corresponding to p'. Using the same argument as in the previous case for the first part, we can prove the first part of the claim for p'.

$p'.trimList_{\mathcal{PH}}[0] = p.trimList_{\mathcal{PH}'}[0]$. Using induction hypothesis, $p.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p)$. Since $succ_{\mathcal{PH}}(p') = succ_{\mathcal{PH}'}(p)$, we have the second part of the claim.

- $o = leave(p) \vee fail(p)$

Note that $\mathcal{P}_{\mathcal{PH}} = \mathcal{P}_{\mathcal{PH}'} - \{p\}$. Let $p'' \in P_{\mathcal{PH}}$ such that $succ_{\mathcal{PH}'}(p'')$=$p$. Then, $succ_{\mathcal{PH}}(p'') = succ_{\mathcal{PH}'}(p) = p_s$.

$\forall p' \in \mathcal{P}_{\mathcal{PH}}$, $p'.trimList_{\mathcal{PH}}$ is same as $p'.trimList_{\mathcal{PH}'}$, except for entries corresponding to p being deleted.

Using the induction hypothesis, $\forall p' \in \mathcal{P}_{\mathcal{PH}'}, \forall i\ 0 \leq i < p'.trimList_{\mathcal{PH}'}.length - 1, p'.trimList_{\mathcal{PH}'}[i] = p \iff p'.trimList_{\mathcal{PH}'}[i+1] = succ_{\mathcal{PH}'}(p) = p_s$ (since $succ$ function is a bijection). Similarly $\forall p' \in \mathcal{P}_{\mathcal{PH}'}, \forall i\ 0 \leq i < p'.trimList_{\mathcal{PH}'}.length - 1, p'.trimList_{\mathcal{PH}'}[i] = p'' \iff p'.trimList_{\mathcal{PH}'}[i+1] = succ_{\mathcal{PH}'}(p'') = p$.

$\forall p' \in \mathcal{P}_{\mathcal{PH}}, \forall i\ 0 \leq i < p'.trimList_{\mathcal{PH}}.length - 1, p'.trimList_{\mathcal{PH}}[i] = p'' \Rightarrow p'.trimList_{\mathcal{PH}}[i+1] = succ_{\mathcal{PH}'}(p) = p_s = succ_{\mathcal{PH}}(p'')$.

$\forall p' \in \mathcal{P}_{\mathcal{PH}}, \forall i\ 0 \leq i < p'.trimList_{\mathcal{PH}}.length - 1, p'.trimList_{\mathcal{PH}}[i] \neq p'' \Rightarrow (\exists j\ p'.trimList_{\mathcal{PH}}[i] = p'.trimList_{\mathcal{PH}'}[j] \wedge p'.trimList_{\mathcal{PH}}[i+1] = p'.trimList_{\mathcal{PH}'}[j+1]\ )$ (only entries corresponding to p are deleted in $p'.trimList_{\mathcal{PH}}$; here $p'.trimList_{\mathcal{PH}'}[j+1] \neq p$ as $p'.trimList_{\mathcal{PH}'}[i] \neq p''$).

Using the induction hypothesis, $p'.trimList_{\mathcal{PH}'}[j+1] = succ_{\mathcal{PH}'}(p'.trimList_{\mathcal{PH}'}[j])$. Since $succ_{\mathcal{PH}'}(p'.trimList_{\mathcal{PH}'}[j]) = succ_{\mathcal{PH}}(p'.trimList_{\mathcal{PH}'}[j])$, we have $p'.trimList_{\mathcal{PH}}[i+1] = succ_{\mathcal{PH}}(p'.trimList_{\mathcal{PH}}[i])$.

We are therefore through with the first claim.

Suppose $p'.trimList_{\mathcal{PH}'}[0] = p$. Using induction hypothesis $p'.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p')$. Therefore $succ_{\mathcal{PH}'}(p') = p$. Since $succ$ is a bijection, $p' = p''$.

Therefore, for $p'.trimList_{\mathcal{PH}'}[0] = p \iff p' = p''$. Using induction hypothesis, $p'.trimList_{\mathcal{PH}'}[1] = succ_{\mathcal{PH}'}(p'.trimList_{\mathcal{PH}'}[0])$. Therefore, $p'.trimList_{\mathcal{PH}'}[1] = succ_{\mathcal{PH}'}(p) = succ_{\mathcal{PH}}(p'')$. Hence, $p''.trimList_{\mathcal{PH}}[0] = p''.trimList_{\mathcal{PH}'}[1] = succ_{\mathcal{PH}}(p'')$.

Now consider $p' \neq p''$. $\forall p' \in \mathcal{P}_{\mathcal{PH}}, p' \neq p'' \Rightarrow p'.trimList_{\mathcal{PH}'}[0] \neq p \Rightarrow p'.trimList_{\mathcal{PH}'}[0] = p'.trimList_{\mathcal{PH}}[0]$. Using the induction hypothesis, $p'.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p')$. Since $succ_{\mathcal{PH}}(p') = succ_{\mathcal{PH}'}(p')$, we have $p'.trimList_{\mathcal{PH}}[0] = succ_{\mathcal{PH}}(p')$.

- $o = Pstab4_i(p, p')$

$succ_{\mathcal{PH}} = succ_{\mathcal{PH}'}$. $\forall p'' \in \mathcal{P}_{\mathcal{PH}}, p'' \neq p, p''.trimList_{\mathcal{PH}} = p''.trimList_{\mathcal{PH}'}$. Using the induction hypothesis, we are through for all $p'' \neq p$.

Now consider peer p. $\forall i\ 0 < i < p.trimList_{\mathcal{PH}}.length, p.trimList_{\mathcal{PH}}[i] = p'.trimList_{\mathcal{PH}'}[i-1]$. $p.trimList_{\mathcal{PH}}[0] = p'$. Using induction hypothesis, $succ_{\mathcal{PH}'}(p'.trimList_{\mathcal{PH}'}[i-1]) = p'.trimList_{\mathcal{PH}'}[i]$. Since $succ_{\mathcal{PH}} = succ_{\mathcal{PH}'}$, $\forall i\ 0 < i < p.trimList_{\mathcal{PH}}.length, p.trimList_{\mathcal{PH}}[i] = succ_{\mathcal{PH}}(p.trimList_{\mathcal{PH}}[i-1])$. Using induction hypothesis again, $p'.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p')$. Therefore, $p.trimList_{\mathcal{PH}}[1] = p'.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}}(p') = succ_{\mathcal{PH}}(p.trimList_{\mathcal{PH}}[0])$. We are therefore through with the first part of the claim.

Using induction hypothesis, $p.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p)$. Therefore, $p.trimList_{\mathcal{PH}}[0] = p.trimList_{\mathcal{PH}'}[0] = succ_{\mathcal{PH}'}(p) = succ_{\mathcal{PH}}(p)$.

■

Corollary 1.1 (Correctness of $getSucc$) *Given* PEPPER Ring History $\mathcal{PH}$, *let* $\mathcal{H} = \Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{PH})$. *Then,* $\forall p, p' \in \mathcal{P}$ ($o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow$ ($\exists o' = PinitGetSucc1_i(p,p') \in O_{\mathcal{H}}\ initGetSucc_i(p,p') \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'$)).

Proof: We need to show that: $\forall p, p' \in \mathcal{P}$ ($o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow$ ($\exists o' \in O_{\mathcal{H}}\ initGetSucc_i(p,p') \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'$)).

From table 1, $o = getSucc_i(p,p') \in O_{\mathcal{PH}} \Rightarrow o' = PinitGetSucc1_i(p,p') \in O_{\mathcal{PH}} \wedge o' \leq_{\mathcal{PH}} o$

From implementation of stabilization and `getSucc` (Algorithms 16 and 21), $o' = PinitGetSucc1_i(p,p') \in O_{\mathcal{PH}} \Rightarrow (p' == \texttt{NULL} \vee (p.succList_{\mathcal{PH}_{o'}}[i] = p' \wedge p.succList_{\mathcal{PH}_o}[i].state == JOINED, \wedge p.succList_{\mathcal{PH}_o}[j].state \neq JOINED, \forall j < i)$. Also, $p' \in \mathcal{P}_{\mathcal{PH}_o} \Rightarrow p' \in \mathcal{P}_{\mathcal{PH}_{o'}}$.

($p' = \texttt{NULL} \vee p' = first$ `JOINED` $successor$) $\wedge p' \in \mathcal{P}_{\mathcal{PH}_{o'}} \Rightarrow p' = p.trimList_{\mathcal{PH}_{o'}}[0]$.

Noting that $PH_{o'}$ is a *PEPPER Ring History* and using the above lemma, $p.trimList_{\mathcal{PH}_{o'}}[0] = succ_{\mathcal{PH}_{o'}}(p)$. Therefore, $p' = succ_{\mathcal{PH}_{o'}}(p)$. Now using equations (3) and (4) we conclude $\forall p, p' \in \mathcal{P}$ ($o = getSucc_i(p,p') \in O_{\mathcal{H}} \wedge p' \in \mathcal{P}_{\mathcal{H}_o} \Rightarrow$ ($\exists o' \in O_{\mathcal{H}}\ initGetSucc_i(p,p') \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'$)).

■

Corollary 1.2 (Correctness of $sendToSucc$) *Given* PEPPER Ring History $\mathcal{PH}$, *let* $\mathcal{H} = \Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{PH})$. *Then,*

1. $\forall p, p' \in \mathcal{P}$ ($o = recvSendToSucc_i(p',p) \in O_{\mathcal{H}} \Rightarrow \exists o'\ initSendToSucc_i(p) \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'$)
2. $\forall p, p' \in \mathcal{P}$ ($o = sendToSucc_i(p,p') \in O_{\mathcal{H}} \Rightarrow$ ($\exists o'\ initSendToSucc_i(p) \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o \wedge succ_{\mathcal{H}_{o'}}(p) = p'$) $\wedge$ ($\forall p'' \in \mathcal{P}\ \nexists insert(p,p'') \in O_{\mathcal{H}} \wedge o' \leq_{\mathcal{H}} insert(p,p'') \leq_{\mathcal{H}} o$))

Proof: `(Proof of 1)`: Let $o = recvSendToSucc_i(p',p) \in O_{\mathcal{PH}}$. Note that $\mathcal{PH}_o$ is a *PEPPER Ring History* and $recvSendToSucc_i(p',p) \in O_{\mathcal{PH}_o}$. From table 1, $recvSendToSucc_i(p',p) \in O_{\mathcal{PH}_o} \Rightarrow initSendToSucc_i(p) \in O_{\mathcal{PH}_o} \wedge initSendToSucc_i(p) \leq_{\mathcal{PH}_o} recvSendToSucc_i(p',p)$.

Let $o_1 = PinitSendToSucc1_i(p,p')$. From $sendToSuccessor$ algorithm (see Algorithm 19) $p' \in \mathcal{P}_{\mathcal{PH}_{o_1}}$. Using the same argument as used in proof of Corollary 1.1, we can prove that $p' = p.trimList_{\mathcal{PH}_{o_1}}[0]$. Using the above lemma, $p.trimList_{\mathcal{PH}_{o_1}}[0] = succ_{\mathcal{PH}_{o_1}}(p)$. Hence, $p' = succ_{\mathcal{PH}_{o_1}}(p)$.

Therefore, $\exists o_1\ initSendToSucc_i(p) \leq_{\mathcal{PH}} o_1 \leq_{\mathcal{PH}} o\ succ_{\mathcal{H}_{o'}}(p) = p'$. Using equations (3) and (4), $\exists o_1\ initSendToSucc_i(p) \leq_{\mathcal{H}} o_1 \leq_{\mathcal{H}} o\ succ_{\mathcal{H}_{o'}}(p) = p'$.

`(Proof of 2)`: Let $o = sendToSucc_i(p',p) \in O_{\mathcal{PH}}$. As shown in (Proof of 1), $\exists o_1\ initSendToSucc_i(p) \leq_{\mathcal{H}} o_1 \leq_{\mathcal{H}} o\ succ_{\mathcal{H}_{o'}}(p) = p'$. Moreover, from table 2, $\forall p'', insert(p,p'')$ cannot occur between o_1 and $sendToSucc(p,p')$ (*duration conflict*).

Therefore, using equations (3) and (4), $\forall p, p' \in \mathcal{P}$ ($o = sendToSucc_i(p',p) \in O_{\mathcal{H}} \Rightarrow$ ($\exists o'\ initSendToSucc_i(p) \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o\ succ_{\mathcal{H}_{o'}}(p) = p'$) $\wedge$ ($\forall p'' \in \mathcal{P}\ \nexists insert(p,p'') \in O_{\mathcal{H}} \wedge o' \leq_{\mathcal{H}} insert(p,p'') \leq_{\mathcal{H}} o$)).

■

Lemma 8 *Given a* PEPPER Ring History $\mathcal{PH} = (O_{\mathcal{PH}}, \leq_{\mathcal{PH}})$, $\forall p, p', p'' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($p \neq p' \wedge o = Pinsert2(p,p') \in O_{\mathcal{PH}} \wedge 0 \leq i < p''.trimList_{\mathcal{PH}_o}.length \wedge p''.trimList_{\mathcal{PH}_o}[i] = p' \Rightarrow p''.trimList_{\mathcal{PH}_o}[i-1] = p$)).

Proof: <`Proof of this lemma uses claims 7 and 10`>.

From table 1, $\exists p'' \in \mathcal{P}\ \exists j \in \mathcal{N}\ Pstab6_j(p'',p,p') \leq_{\mathcal{H}} Pinsert2(p,p')$

Using claim 7, $Pstab6_j(p'',p,p') \in O_{\mathcal{H}} \Rightarrow PinitInsert2(p,p') \leq_{\mathcal{H}} Pstab6_j(p'',p,p')$.

Therefore, $o = Pinsert2(p,p') \in O_{\mathcal{PH}} \Rightarrow o' = PinitInsert2(p,p') \leq_{\mathcal{PH}} o$.

Given $p''.trimList_{\mathcal{PH}_o}[i] = p'$, $\exists j\ j > i \wedge p''.succList_{\mathcal{PH}_o}[j] = p'$.

We also have that $o = Pinsert2(p,p') \Rightarrow live_{\mathcal{PH}_o}(p)$.

Using claim 10, we conclude that $p''.succList_{\mathcal{PH}_o}[j-1] = p$. Therefore, $p''.trimList_{\mathcal{PH}_o}[i-1] = p$ and hence we are through.

■

Claim 10 *Given a* PEPPER history $\mathcal{PH} = (O_{\mathcal{PH}}, \leq_{\mathcal{PH}})$, $\forall p, p', p'' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall o' \in O_{\mathcal{PH}}$ ($p \neq p' \wedge o = PinitInsert2(p,p') \in O_{\mathcal{PH}} \wedge o \leq_{\mathcal{PH}} o' \wedge (\nexists o''\ o'' = Pinsert2(p,p') \in O_{\mathcal{PH}} \wedge o \leq_{\mathcal{PH}} o'' \leq_{\mathcal{PH}} o') \wedge 0 \leq i < p''.succList_{\mathcal{PH}_{o'}}.length \wedge p''.succList_{\mathcal{PH}_{o'}}[i].peerid = p' \wedge live_{\mathcal{PH}_{o'}}(p) \Rightarrow p''.succList_{\mathcal{PH}_{o'}}[i-1].peerid = p$))).

Proof: <`Proof of this lemma uses claims 11`>

By induction on the number of operations in PEPPER Ring History $\mathcal{PH}$.

Base Case: $\mathcal{PH} = \{initRing(p), PinitRing(p), insert(p,p)\}$ (This is the shortest allowed *PEPPER Ring History*). In this case, the claim is trivially true.

Induction Hypothesis: Let's assume that the claim holds for any *PEPPER Ring History* $\mathcal{PH}'$ such that $|O_{\mathcal{PH}'}| = k$.

Induction Step: We show that the claim holds for any *PEPPER Ring History* $\mathcal{PH}$ such that $|O_{\mathcal{PH}}| = k+1$.

Let $op \in O_{\mathcal{PH}}, op \neq insert(p,p)$ for any $p \in \mathcal{P}$, such that $\nexists o' \in O_{\mathcal{PH}}$ ($op \leq_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - op$.

Using claim 6, $\mathcal{PH}' = \Pi_{O'}(\mathcal{PH})$ is also a *PEPPER Ring History*.

$\mathcal{PH}'$ is a *PEPPER Ring History* and $|O_{\mathcal{PH}'}| = k$. Moreover $o' \in O_{\mathcal{PH}} \wedge o' \neq op \Rightarrow o' \in O_{\mathcal{PH}'}$. Therefore, using the induction hypothesis, claim holds for *PEPPER Ring History* $\mathcal{PH}'$.

$\forall p \in \mathcal{P}$ ($\forall o' \neq op$ ($p.succList_{\mathcal{PH}_{o'}} = p.succList_{\mathcal{PH}'_{o'}}$)). Therefore, in *PEPPER Ring History* $\mathcal{PH}$, the claim holds for all $o' \in O_{\mathcal{PH}}$ except possibly for $o' = op$.

If op does not modify $p.succList$, $p.succList_{\mathcal{PH}_{op}} = p.succList_{\mathcal{PH}'}$. Using the induction hypothesis, we are through in this case. We now consider only operations that modify $p.succList$ (see tables 3).

Now let $o = PinitInsert2(p, p') \in O_{\mathcal{PH}'}$.

- $op = PinitInsert2(p, p'')$:

  Note that $o = PinitInsert2(p, p') \in O_{\mathcal{PH}'}$.

  Using table 1 $\forall i \in \mathcal{N}$ ($o = Pinsert2_i(p, p') \in O_{\mathcal{PH}} \Rightarrow \exists j \in \mathcal{N}\ Pstab6_j(p'', p, p') \leq_{\mathcal{PH}} Pinsert2_i(p, p')$).

  Using claim 7, $Pstab6_j(p'', p, p') \in O_{\mathcal{PH}} \Rightarrow PinitInsert2(p, p') \leq_{\mathcal{PH}} Pstab6_j(p'', p, p')$.

  Using API restriction 3(b), $p'' \neq p'$.

  From implementation of $PinitInsert2(p, p'')$, $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}'}.length \wedge p'' \neq p \Rightarrow p''.succList_{\mathcal{PH}'} = p''.succList_{\mathcal{PH}}$)). Moreover, from the implementation of $PinitInsert2(p, p')$, $\forall i$ ($0 < i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p.succList_{\mathcal{PH}'}[i-1]$) $\wedge\ p.succList_{\mathcal{PH}}[0].peerid = p'' \neq p' \wedge p.succList_{\mathcal{PH}}[0].state =$ `JOINING`.

  Suppose, $p.succList_{\mathcal{PH}}[1] = p'$. Then $p.succList_{\mathcal{PH}'}[0].peerid = p'$. Using claim 11, $p.succList_{\mathcal{PH}'}[0].state =$ `JOINING`. Therefore $p.succList_{\mathcal{PH}}[1].state =$ `JOINING`. Using claim 8, $p.succList_{\mathcal{PH}}[0].state =$ `JOINING` $\Rightarrow p.succList_{\mathcal{PH}}[1].state \neq$ `JOINING`. A contradiction.

  Therefore $p.succList_{\mathcal{PH}}[1] \neq p'$. Also, $live_{\mathcal{PH}}(p) \Rightarrow live_{\mathcal{PH}'}(p)$. Using the induction hypothesis, we conclude that the claim holds in this case.

  If $o_1 = PinitInsert2(p, p'')$, only possible o'_1 to consider such that $o_1 \leq_{\mathcal{PH}} o'_1 \wedge (\nexists o''\ o'' = Pinsert2(p, p_1) \in O_{\mathcal{PH}} \wedge o_1 \leq_{\mathcal{PH}} o'' \leq_{\mathcal{PH}} o'_1)$ is $o'_1 = o_1 = op$. Using claim 12, $p''.succList_{\mathcal{PH}}[i].peerid = p''$ iff $p'' = p \wedge i = 0$. Since $p.succList_{\mathcal{PH}}[-1].peerid = p$, we are through in this case.

- $op = Pinsert2(p, p'')$

  $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}}.length \Rightarrow p''.succList_{\mathcal{PH}'}[i].peerid = p''.succList_{\mathcal{PH}}[i].peerid$)).

  Using the induction hypothesis, we are through.

- $op = Pjoin(p, p'')$

  From table 1, $Pjoin(p, p'') \leq_{\mathcal{PH}} inserted(p)$.

  From API restriction 3(c), $\forall p' \in \mathcal{P}$ ($inserted(p) \leq_{\mathcal{PH}} PinitInsert2(p, p')$).

  Therefore $Pjoin(p, p'') \leq_{\mathcal{PH}} PinitInsert2(p, p')$), contradicting $o = PinitInsert2(p, p') \in O_{\mathcal{PH}}$. Therefore $op \neq Pjoin(p, p'')$.

- $op = Pping3_i(p, p'')$, $Pping6_i(p, p'')$

  If $\neg live_{\mathcal{PH}}(p)$ we are through.

  Assume $live_{\mathcal{PH}}(p)$. Consider p'', i such that $p''.succList_{\mathcal{PH}}[i] = p'$. Then, $\exists j\ p''.succList_{\mathcal{PH}'}[j] = p'$. Using induction hypothesis, $p''.succList_{\mathcal{PH}'}[j-1] = p$. Since $live_{\mathcal{PH}}(p)$, $p'' \neq p$. Therefore, $p''.succList_{\mathcal{PH}}[i-1] = p$.

- $op = Pstab4_i(p_1, p'')$

  $\forall p'' \in \mathcal{P}$ ($p'' \neq p_1 \Rightarrow p''.succList_{\mathcal{PH}} = p''.succList_{\mathcal{PH}'}$). Using induction hypothesis, we are through in this case.

  Moreover from Algorithms 17 and 18, $\exists j \in \mathcal{N}$ $\forall i \in \mathcal{N}$ (($0 \leq j \leq i < p_1.succList_{\mathcal{PH}}.length \Rightarrow p_1.succList_{\mathcal{PH}}[i].peerid = p''.succList_{\mathcal{PH}'}[i-j-1].peerid$) $\wedge$ ($0 \leq i < j \Rightarrow p_1.succList_{\mathcal{PH}}[i].peerid = p_1.succList_{\mathcal{PH}'}[i].peerid$)).

  Hence, by induction hypothesis, we are through in this case.

Suppose $\nexists o = PinitInsert2(p, p') \in O_{\mathcal{PH}'}$. Only non-trivial case to consider is $op = PinitInsert2(p, p'')$ for some $p'' \in \mathcal{P}$. From implementation of $PinitInsert2(p, p'')$, $\forall i \in \mathcal{N}\ p.succList_{\mathcal{PH}_{op}}[i].peerid = p'' \Rightarrow i = 0$. Also, $p.succList_{\mathcal{PH}_{op}}[-1].peerid = p$. Hence the claim follows.

■

Claim 11 *Given a* PEPPER Ring History $\mathcal{PH} = (O_{\mathcal{PH}}, \leq_{\mathcal{PH}})$, $\forall p, p', p'' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall o' \in O_{\mathcal{PH}}$ ($p \neq p' \wedge o = PinitInsert2(p, p') \in O_{\mathcal{PH}} \wedge o \leq_{\mathcal{PH}} o' \wedge (\nexists o''\ o'' = Pinsert2(p, p') \in O_{\mathcal{PH}} \wedge o \leq_{\mathcal{PH}} o'' \leq_{\mathcal{PH}} o') \wedge 0 \leq i < p''.succList_{\mathcal{PH}_{o'}}.length \wedge p''.succList_{\mathcal{PH}_{o'}}[i].peerid = p' \Rightarrow p''.succList_{\mathcal{PH}_{o'}}[i].state =$ `JOINING`))).

Proof: <`Proof of this lemma uses claim 12`>.

By induction on the number of operations in PEPPER Ring History $\mathcal{PH}$.

Base Case: $\mathcal{PH} = \{initRing(p), PinitRing(p), insert(p, p)\}$ (This is the shortest allowed *PEPPER Ring History*). In this case, the claim is trivially true.

Induction Hypothesis: Let's assume that the claim holds for any *PEPPER Ring History* $\mathcal{PH}'$ such that $|O_{\mathcal{PH}'}| = k$.

Induction Step: We show that the claim holds for any *PEPPER Ring History* $\mathcal{PH}$ such that $|O_{\mathcal{PH}}| = k + 1$.

Let $op \in O_{\mathcal{PH}}, op \neq insert(p, p)$ for any $p \in \mathcal{P}$, such that $\nexists o' \in O_{\mathcal{PH}}$ ($op \leq_{\mathcal{PH}} o'$). Let $O' = O_{\mathcal{PH}} - op$. Using claim 6, $\mathcal{PH}' = \Pi_{O'}(\mathcal{PH})$ is also a *PEPPER Ring History*.

$\mathcal{PH}'$ is a *PEPPER Ring History* and $|O_{\mathcal{PH}'}| = k$. Moreover $o' \in O_{\mathcal{PH}} \wedge o' \neq op \Rightarrow o' \in O_{\mathcal{PH}'}$. Therefore, using the induction hypothesis, claim holds for *PEPPER Ring History* $\mathcal{PH}'$.

$\forall p \in \mathcal{P}$ ($\forall o' \neq op$ ($p.succList_{\mathcal{PH}_{o'}} = p.succList_{\mathcal{PH}'_{o'}}$)). Therefore, in *PEPPER Ring History* $\mathcal{PH}$, the claim holds for all $o' \in O_{\mathcal{PH}}$ except possibly for $o' = op$.

If op does not modify $p.succList$, $p.succList_{\mathcal{PH}_{op}} = p.succList_{\mathcal{PH}'}$. Using the induction hypothesis, we are through in this case. We now consider only operations that modify $p.succList$ (see tables 3).

Now let $o = PinitInsert2(p, p') \in O_{\mathcal{PH}'}$.

- $op = PinitInsert2(p, p''')$:

  Using table 1 $\forall i \in \mathcal{N}$ ($o = Pinsert2_i(p, p') \in O_{\mathcal{PH}} \Rightarrow \exists j \in \mathcal{N}\ Pstab6_j(p'', p, p') \leq_{\mathcal{PH}} Pinsert2_i(p, p')$).

  Using claim 7, $Pstab6_j(p'', p, p') \in O_{\mathcal{PH}} \Rightarrow PinitInsert2(p, p') \leq_{\mathcal{PH}} Pstab6_j(p'', p, p')$.

  Using API restriction 3(b), $p''' \neq p'$.

  Using claim 12, when $op = PinitInsert2(p, p''') = o_1$, only possible $o'_1 = o_1$. From implementation of $PinitInsert2(p, p''')$, $p.succList_{\mathcal{PH}_{o'_1}}[0].peerid = p''' \wedge p.succList_{\mathcal{PH}_{o'_1}}[0].state =$ `JOINING` and hence the claim follows.

  Now when o = PinitInsert2(p, p'), only new o' to consider is op. From implementation of $PinitInsert2(p, p''')$, $\forall p'' \in \mathcal{P}$ ($\forall i$ ($0 \leq i < p''.succList_{\mathcal{PH}'}.length \wedge p'' \neq p \Rightarrow p''.succList_{\mathcal{PH}'} = p''.succList_{\mathcal{PH}}$)). Moreover, from the implementation of $PinitInsert2(p, p')$, $\forall i$ ($0 < i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p.succList_{\mathcal{PH}'}[i-1]$) $\wedge$ $p.succList_{\mathcal{PH}}[0] = p''' \neq p'$.

  Therefore, we are through by the induction hypothesis.

- $op = Pinsert2(p, p''')$

  There are no new o' to consider in this case. The only candidate $o' = op$ does not satisfy the constriant $\forall p_1 \in \mathcal{P}$ ($\nexists o''\ o'' = Pinsert2(p, p_1) \in O_{\mathcal{PH}} \wedge o \leq_{\mathcal{PH}} o'' \leq_{\mathcal{PH}} o'$).

- $op = Pjoin(p, p''')$

  From table 1, $Pjoin(p, p''') \leq_{\mathcal{PH}} inserted(p)$.

  From API restriction 3(c), $\forall p' \in \mathcal{P}$ ($inserted(p) \leq_{\mathcal{PH}} PinitInsert2(p, p')$).

  Therefore $Pjoin(p, p''') \leq_{\mathcal{PH}} PinitInsert2(p, p')$), contradicting $o = PinitInsert2(p, p') \in O_{\mathcal{PH}}$. Therefore $op \neq Pjoin(p, p''')$.

- $Pping3_i(p, p''')$, $Pping6_i(p, p''')$

  $\forall p'' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($p''.succList_{\mathcal{PH}}[i].peerid = p' \Rightarrow \exists j \in \mathcal{N}$ ($p''.succList_{\mathcal{PH}'}[j].peerid = p'$))).

  Therefore, using the induction hypothesis, we are through.

- $op = Pstab4_i(p, p''')$

  $\forall p'' \in \mathcal{P}$ ($p'' \neq p \Rightarrow p''.succList_{\mathcal{PH}} = p''.succList_{\mathcal{PH}'}$). Using induction hypothesis, we are through in this case.

  For $p'' = p$, from Algorithms 17 and 18, we have that $\exists j \in \mathcal{N}\ \forall i \in \mathcal{N}$ (($0 \leq j < i < p.succList_{\mathcal{PH}}.length \Rightarrow p.succList_{\mathcal{PH}}[i] = p'''.succList_{\mathcal{PH}'}[i-j-1]$) $\wedge$ ($0 \leq i < j \Rightarrow p.succList_{\mathcal{PH}}[i] = p.succList_{\mathcal{PH}'}[i]$) $\wedge$ ($p.succList_{\mathcal{PH}}[j].peerid = p.succList_{\mathcal{PH}'}[j].peerid = p'''$ $p.succList_{\mathcal{PH}}[j].state \neq$ `JOINING`)).

  If we prove that $p''' \neq p'$, from induction hypothesis, the claim follows.

  Since $\nexists o'' = Pinsert2(p, p') \in O_{\mathcal{PH}} \wedge o <_{\mathcal{PH}} o'' <_{\mathcal{PH}} o'$, from Table 1, $\nexists o''' = Pjoin(p', p) \in O_{\mathcal{PH}} \wedge o <_{\mathcal{PH}} o''' <_{\mathcal{PH}} o'$. The only way p' could process a stabilization message, so we can have operation $Pstab4_i(p, p')$ is if $Pjoin(p', p) <_{\mathcal{PH}} Pstab4_i(p, p')$. Since this is not the case, we have that $p''' \neq p'$.

  Therefore $p''' \neq p'$. Hence, we are through by induction hypothesis.

Suppose $\nexists o = PinitInsert2(p, p') \in O_{\mathcal{PH}'}$. Only non-trivial case to consider is $op = PinitInsert2(p, p''')$ for some $p''' \in \mathcal{P}$. From implementation of $PinitInsert2(p, p''')$, $p.succList_{\mathcal{PH}_{op}}[0].peerid = p''' \wedge p.succList_{\mathcal{PH}_{op}}[0].state =$ `JOINING` and hence the claim follows.

■

Claim 12 *Given a* PEPPER Ring History $\mathcal{PH} = (O_{\mathcal{PH}}, \leq_{\mathcal{PH}})$, $\forall p, p', p'' \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall o' \in O_{\mathcal{PH}}$ ($p \neq p' \wedge o = PinitInsert2(p, p') \in O_{\mathcal{PH}} \wedge o' <_{\mathcal{PH}} o \wedge 0 \leq i < p''.succList_{\mathcal{PH}_{o'}}.length - 1 \Rightarrow p''.succList_{\mathcal{PH}_{o'}}[i].peerid \neq p'$))).

Proof: $o = PinitInsert2(p, p')$ is the operation which first introduces p' into $p.succList$. None of the peers in the ring know about p' before o. Hence, the result holds.

■

12 Data Store Correctness

In this section, we first present the *API Data Store History*. We then present the *PEPPER Data Store History* which implements this API. We then move onto showing that the PEPPER Data Store satisfies all the important correctness requirements of the API Data Store History.

12.1 API Data Store History

In this section we present the API methods supported by the Data Store. We present the operations associated with initiation and completion of the API methods, as well as the operations associated with the events exposed by the Data Store. We then define an *API Data Store History* using these operations.

The following methods are part of the Data Store API:

- p.`initFirstPeer(`p`)`
- p.`initScanRange(`$lb, ub, handlerID, params$`)`
- p.`initGetLocalItems()`
- p.`initInsertLocalItem(`j`)`
- p.`initDeleteLocalItem(`j`)`
- p.`initInsertItem(`j`)`
- p.`initDeleteItem(`j`)`
- p.`initSendToSucc(`msg`)`
- p.`initGetSucc()`
- p.`fail()`

We now explain what each API method does and specify the initiation and completion operations associated with the method.

- p.`initFirstPeer(`p`)`: This method is used to insert the first peer into the system. $initFirstPeer(p)$ is the operation associated with the invocation of p.`initFirstPeer(`p`)`. $firstPeer(p)$ is the operation used to signal the completion of this API method.
- p.`initScanRange(`$lb, ub, handlerID, params$`)`: This method causes handlers with id $handlerID$ to be invoked with parameters $params$ at all peers whose range overlaps with $[lb, ub]$. $initScanRange_i(p, lb, ub), i, lb, ub \in \mathcal{N}$ [2] is the operation associated with the invocation of p.`initScanRange(`$lb, ub, handlerID, params$`)`. $scanRange_{ik}(p', p, r'), i, k \in \mathcal{N}, r' \subseteq [lb, ub]$, is the operation used to signal the API operation p.`initScanRange(`$lb, ub, handlerID, params$`)` reaching peer p'. $doneScanRange_i(p, lb, ub), i, lb, ub \in \mathcal{N}$ is the operation used to signal the end of the above API operation at peer p.
- p.`initGetLocalItems()`: This method returns the $items$ in p's data store. $initGetLocalItems_i(p), i \in \mathcal{N}$, is the operation associated with the invocation of p.`initGetLocalItems()`. $getLocalItems_i(p, items), i \in \mathcal{N}$, is the operation used to signal the end of this API operation.
- p.`initInsertItem(`j`)`: This API method is used to insert item j into the system. $initInsertItem(p, j)$ is the operation associated with the invocation of p.`initInsertItem(`j`)`. $insertItem(p, j)$ is the operation used to signal the end of this API operation at peer p. $insertedItem(p', p, j)$ is used to signal the insert of item j at peer p' and initiated by peer p.
- p.`initDeleteItem(`j`)`: This API methods is used to delete items j from the system. $iniDeleteItem(p, j)$ is the operation associated with the invocation of p.`initDeleteItem(`j`)`. $deleteItem(p, j)$ is the operation used to signal the end of this API operation at peer p. $deletedItem(p', p, j)$ is used to signal the delete of item j at peer p' and initiated by peer p.
- p.`fail()`: $p.fail()$ is considered part of the API to capture peer failures. It is not a method which can be called by higher layers. We use the operation $fail(p)$ to denote the failure of a peer p.
- p.`initSendToSucc(`msg`)`, p.`initGetSucc()`: These are ring API methods which are also exposed at the Data Store. These API methods are excluded in the current discussion of API Data Store History in the interest of space.

In addition, the following events are thrown up by the Data Store.

- `DSINFOFORSUCCEVENT`: This is a synchronous event. $initDSInfoForSuccEvent(p, p')$ is the operation that is used to denote the initiation of the event at peer p which gathers data from higher layers that needs to be sent to the new peer p'. $dsInfoForSuccEvent(p, p')$ denotes the completion of the event.
- `DSINFOFROMPREDEVENT`: $dsInfoFromPredEvent(p)$ is the operation that denotes the event at peer p' which causes initialization of higher layers.
- `LEAVEEVENT`: $leaveEvent(p)$ is the operation used to denote the leave event at peer p which is thrown before p leaves the ring (because of merge).
- `RANGECHANGEEVENT`: This is a synchronous event. $initRangeChangeEvent_i(p, r, b)$ denotes the initiation of the range change event at peer p with new range r. Flag b is 1 when range change

[2] For ease of exposition we assume an integer domain for *search key values*. In general, the domain could be any arbitrary domain with a total order.

is because of a new predecessor; 0 otherwise. $rangeChangeEvent_i(p, r, b)$ denotes the completion of the event.

- NEWSUCCEVENT: This is an event thrown by the ring layer. The Data Store also throws up the same event. This event is not included in the current discussion of API Data Store History in the interest of space.

Before we define an API Data Store History, we first present some useful notation.

Notation($\mathcal{T}$): We use $\mathcal{T}$ to denote the set of all possible items.

Notation($range_{\mathcal{H}}(p)$): Given a history $\mathcal{H}$, we define the range of peer p, $range_{\mathcal{H}}(p)$, as follows:

- $range_{\mathcal{H}}(p) = r \iff o = rangeChangeEvent(p, r, b) \in O_{\mathcal{H}} \wedge (\nexists o' \in O_{\mathcal{H}} \ (\ o' = rangeChangeEvent(p, r', b') \wedge o \leq_{\mathcal{H}} o' \wedge o \neq o' \ ))$
- $\nexists o = rangeChangeEvent(p, r, b) \in O_{\mathcal{H}} \Rightarrow range_{\mathcal{H}}(p) = \phi$

Notation($range_1 \bowtie range_2$): $range_1 \bowtie range_2$ is true iff $range_1 \cap range_2 \neq \emptyset$.

Notation($scanOps(i)$): We use $scanOps(i)$ to denote the set of $scanRange_i(p, p_1, r)$ operations associated with the i^{th} invocation of scanRange.

Notation($rangeSet(i)$): We use $rangeSet(i) = \{r | \exists p_1, p_2$ $scanRange_i(p_1, p_2, r) \in scanOps(i)\}$ to denote the set of ranges reached by scanRange.

Notation($\mathcal{O}_d(p)$): $\mathcal{O}_d(p) = \{initFirstPeer(p), firstPeer(p),$ $initScanRange_i(p, lb, ub), doneScanRange_i(p, lb, ub),$ $initInsertItem(p, j), insertItem(p, j), leaveEvent(p),$ $initRangeChangeEvent_i(p, r, b),$ $rangeChangeEvent_i(p, r, b), initDeleteItem(p, j),$ $deleteItem(p, j), initgetLocalItems_i(p),$ $getLocalItems_i(p, j[]), fail(p) | i, lb, ub, b \in \mathcal{N}, j \in \mathcal{T}\}$ denotes all operations at peer p involving only peer p.

Notation($\mathcal{O}_d(p, p')$): $\mathcal{O}_d(p, p') = \{scanRange_{ik}(p), insertedItem(p', p, j),$ $deletedItem(p', p, j), dsInfoForSuccEvent(p, p'),$ $initDSInfoForSuccEvent(p, p'),$ $dsInfoFromPredEvent(p), newSuccEvent_i(p, p') | i, k \in \mathcal{N}, j \in \mathcal{T}\}$ denotes all operations at peer p involving only peers p and p'.

Notation($\mathcal{O}_d(\mathcal{P})$): Given a set of peers $\mathcal{P}$, the set of operations under consideration are $\mathcal{O}_d(\mathcal{P}) = \{initFirstPeer(p),$ $firstPeer(p), initScanRange_i(p, lb, ub),$ $doneScanRange_i(p, lb, ub), scanRange_{ik}(p),$ $initGetLocalItems_i(p), getLocalItems_i(p, j[]),$ $initInsertItem(p, j), insertItem(p, j),$ $insertedItem(p', p, j), initDeleteItem(p, j), deleteItem(p, j),$ $deletedItem(p', p, j), dsInfoForSuccEvent(p, p'),$ $initDSInfoForSuccEvent(p, p'),$ $dsInfoFromPredEvent(p), leaveEvent(p),$ $initRangeChangeEvent_i(p, r, b),$ $rangeChangeEvent_i(p, r, b), newSuccEvent_i(p, p'),$ $fail(p) | p, p' \in \mathcal{P}, i, k, lb, ub, b \in \mathcal{N}, j \in \mathcal{T}\}$.

Definition 13 ($items_{\mathcal{H}}$) *Let $\mathcal{H}$ be a history such that $O_{\mathcal{H}} \subseteq \mathcal{O}_d(\mathcal{P})$. We define $items_{\mathcal{H}}(p)$, the items with a peer p in $\mathcal{H}$, by inducting on the operations in $O_{\mathcal{H}}$ as follows:*

1. *We use $items_{\mathcal{H}}(\phi)$ to denote the items which have been inserted into the system but are not currently with any peer. Note that we are not interested in items with failed peers as these items cannot necessarily be retrieved from the system.*
2. Base case*: $\mathcal{H} = \{initFirstPeer(p), firstPeer(p)\}$. In this case, $items_{\mathcal{H}}(p) = \{\}$ and $items_{\mathcal{H}}(\phi) = \{\}$.*
3. Induction Hypothesis*: We assume that $items_{\mathcal{H}'}$ is defined for all $\mathcal{H}'$ such that $|O_{\mathcal{H}'}| = k$.*
4. Induction Step*: Now consider $\mathcal{H}$ such that $|O_{\mathcal{H}}| = k + 1$. Consider an op $o \in O_{\mathcal{H}}$ such that $\nexists o' \in \mathcal{H}$ $o <_{\mathcal{H}} o'$ (Such operation o exists). $\mathcal{H}' = (O_{\mathcal{H}} - \{o\}, \leq_{\mathcal{H}'})$, where $o_1 \leq_{\mathcal{H}'} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} - \{o\} \wedge o_1 \leq_{\mathcal{H}} o_2$, is also a history. Since $|\mathcal{H}'| = k$, by induction hypothesis, $items_{\mathcal{H}'}$ is defined.*

   We now consider different possibilities for operation o.

   - $o = insertedItem(p, p', j)$*: $\forall p_1 \neq p \ (items_{\mathcal{H}}(p_1) = items_{\mathcal{H}'}(p_1))$. $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) \cup \{j\}$. $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi)$*
   - $o = deletedItem(p, p', j)$*: $\forall p_1 \neq p \ (items_{\mathcal{H}}(p_1) = items_{\mathcal{H}'}(p_1))$. $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) - \{j\}$. $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi)$*
   - $o = insertLocalItem(p, j)$*: $\forall p_1 \neq p \ (items_{\mathcal{H}}(p_1) = items_{\mathcal{H}'}(p_1))$. $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) \cup \{j\}$. $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi)$*
   - $o = deleteLocalItem(p, j)$*: $\forall p_1 \neq p \ (items_{\mathcal{H}}(p_1) = items_{\mathcal{H}'}(p_1))$. $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) - \{j\}$. $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi)$*
   - $o = rangeChangeEvent(p, r, b)$*: $\forall p_1 \neq p \ (items_{\mathcal{H}}(p_1) = items_{\mathcal{H}'}(p_1))$.*

     (We assume here that $r.low = range_{\mathcal{H}'}(p).low || r.high = range_{\mathcal{H}'}(p).high$. This is one of the requirements of an API Data Store History.)

     If $r \subseteq range_{\mathcal{H}'}(p)$ then $i \in items_{\mathcal{H}}(p) \iff i \in items_{\mathcal{H}'}(p) \wedge i.skv \in r$ and $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi) \cup items_{\mathcal{H}'}(p) - items_{\mathcal{H}}(p)$.

     If $r \supset range_{\mathcal{H}'}(p)$ then $i \in items_{\mathcal{H}}(p) \iff i \in items_{\mathcal{H}'}(p) \vee (i \in items_{\mathcal{H}'}(\phi) \wedge i.skv \in r)$ and $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi) - items_{\mathcal{H}}(p)$.

- $o = fail(p)$: $\forall p_1 \neq p\ (items_{\mathcal{H}}(p_1) = items_{\mathcal{H}'}(p_1)\ ).\ items_{\mathcal{H}}(p) = \{\}.\ items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi) - \{i | i \in items_{\mathcal{H}'}(\phi) \wedge i.skv \in range_{\mathcal{H}'(p)}\}$ *(We are not interested in items with the failed peer or items in transit which are supposed to be with the failed peer).*
- *For any other* o, $\forall p\ (items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p)\ )$ *and* $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi)$.

Notation: ($O_{I\mathcal{H}}$): We use $O_{I\mathcal{H}}$ to denote the subset of operations in $O_{\mathcal{H}}$ that affect $items_{\mathcal{H}}$. In particular, $O_{I\mathcal{H}}(p) = \{insertedItem(p, p', j), deletedItem(p, p', j), insertLocalItem(p, j), deleteLocalItem(p, j), rangeChangeEvent(p, r, b), fail(p)\}$ is the set of operations that affect $items_{\mathcal{H}}(p)$.

We define an API Data Store History as follows:

Definition 14 (API Data Store History) *Given a set of peers* $\mathcal{P}$, $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ *is an* API Data Store History *iff*

1. $\mathcal{H}$ *is a history.*
2. $O_{\mathcal{H}} \subseteq \mathcal{O}_d(\mathcal{P})$
3. API restrictions*: These are restrictions on how the Data Store API can be used.*
   - *(a)* $\exists p \in \mathcal{P}\ (\ initFirstPeer(p) \in O_{\mathcal{H}} \wedge firstPeer(p) \in O_{\mathcal{H}} \wedge \forall p' \in \mathcal{P}\ (\ initFirstPeer(p') \in O_{\mathcal{H}} \Rightarrow p = p'\ )\ )$.
     (There exists a unique peer p *which starts off the system.)*
   - *(b)* $\forall p \in \mathcal{P}\ (\ op(p) \in \{initScanRange_i(p, lb, ub), initInsertItem(p, j), initDeleteItem(p, j), initGetLocalItems_i(p), initInsertLocalItem(p, j), initDeleteLocalItem(p, j), i, lb, ub \in \mathcal{N}, j \in \mathcal{T}\}\ ) \wedge op(p) \in O_{\mathcal{H}} \Rightarrow dsInfoFromPredEvent(p) \leq_{\mathcal{H}} op(p)\ )\ )$
     (All API operations other than $initFirstPeer$ *can be initiated only after* DSINFOFROMPREDEVENT*)*
   - *(c)* $\forall p \in \mathcal{P}\ (\ leaveEvent(p) \in O_{\mathcal{H}} \Rightarrow\ (\ op(p) \in \{initFirstPeer(p), initScanRange_i(p, lb, ub), initInsertItem(p, j), initDeleteItem(p, j), initGetLocalItems_i(p), initInsertLocalItem(p, j), initDeleteLocalItem(p, j), i, lb, ub \in \mathcal{N}, j \in \mathcal{T}\}\ ) \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{H}} leaveEvent(p)\ )$
     (No API operation can be initiated after LEAVEEVENT*)*
   - *(d)* $\forall p \in \mathcal{P}\ (\ fail(p) \in O_{\mathcal{H}} \Rightarrow\ (\ op(p) \in \{initFirstPeer(p), initScanRange_i(p, lb, ub), initInsertItem(p, j), initDeleteItem(p, j), initGetLocalItems_i(p), initInsertLocalItem(p, j), initDeleteLocalItem(p, j), i, lb, ub \in \mathcal{N}, j \in \mathcal{T}\}\ ) \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{H}} fail(p)\ )$
     (All operations on peer p *are initiated before* $fail(p)$*)*
   - *(e)* $\forall p, p' \in \mathcal{P}\ (\ dsInfoForSuccEvent(p, p') \in O_{\mathcal{H}} \Rightarrow initDSInfoForSuccEvent(p, p') \leq_{\mathcal{H}} dsInfoForSuccEvent(p, p')\ )$.
     (Any $dsInfoForSuccEvent(p, p')$ *should be preceded by an* $initDSInfoForSuccEvent(p, p')$ *operation)*
   - *(f)* $\forall p \in \mathcal{P}\ (\ \forall i\ (\ \forall r\ (\ \forall b\ (\ rangeChangeEvent_i(p, r, b) \in O_{\mathcal{H}} \Rightarrow initRangeChangeEvent_i(p, r, b) \leq_{\mathcal{H}} rangeChangeEvent_i(p, r, b)\ )\ )\ )\ )$.
     (Any $rangeChangeEvent_i(p, r, b)$ *should be preceded by an* $initRangeChangeEvent_i(p, r, b)$ *operation.)*
   - *(g)* $\forall p, p' \in \mathcal{P}\ (\ \forall j \in \mathcal{T}\ (\ initInsertItem(p, j) \in O_{\mathcal{H}} \wedge initInsertItem(p', j) \in O_{\mathcal{H}} \Rightarrow p = p'\ )\ )$
     (Insert of item $j \in \mathcal{T}$ *is tried at most once.)*
   - *(h)* $\forall p, p' \in \mathcal{P}\ (\ \forall j \in \mathcal{T}\ (\ initDeleteItem(p, j) \in O_{\mathcal{H}} \wedge initDeleteItem(p', j) \in O_{\mathcal{H}} \Rightarrow p = p'\ )\ )$
     (Delete of item $j \in \mathcal{T}$ *is tried at most once.)*
4. Semantic Requirements *These are restrictions on the semantics of the API methods.*
   - *(a)* $\forall p \in \mathcal{P}\ (\ fail(p) \in O_{\mathcal{H}} \Rightarrow\ (\ \forall op(p) \in \mathcal{O}_d(p)\ (\ op(p) \in O_{\mathcal{H}} \wedge op(p) \neq fail(p) \Rightarrow op(p) \leq_{\mathcal{H}} fail(p)\ )\ ) \wedge (\ \forall p' \in \mathcal{P}, \forall op(p, p') \in \mathcal{O}_d(p, p')\ (\ op(p, p') \in O_{\mathcal{H}} \Rightarrow op(p, p') \leq_{\mathcal{H}} fail(p)\ )\ )\ )$.
     (Any operation at p *other than fail* happened before $fail(p)$*.)*
   - *(b)* $\forall p, p' \in \mathcal{P}\ (\ \forall i, k \in \mathcal{N}\ (\ \forall r' \in \mathcal{N} \times \mathcal{N}\ (\ scanRange_{ik}(p', p, r') \in O_{\mathcal{H}} \Rightarrow \exists lb, ub \in \mathcal{N}\ (\ initScanRange_i(p, lb, ub) \in O_{\mathcal{H}} \wedge initScanRange_i(p, lb, ub) \leq_{\mathcal{H}} scanRange_{ik}(p', p, r')\ )\ )\ )\ )$.
     (A scanRange operation should be initiated before it is completed.)
   - *(c)* $\forall p \in \mathcal{P}\ (\ \forall i, lb, ub \in \mathcal{N}\ (\ doneScanRange_i(p, lb, ub) \in O_{\mathcal{H}} \Rightarrow initScanRange_i(p, lb, ub) \leq_{\mathcal{H}} doneScanRange_i(p, lb, ub)\ )\ )$.
     (A scanRange operation should be initiated before it is completed.)
   - *(d)* $\forall p \in \mathcal{P}\ (\ \forall j \in \mathcal{T}\ (\ insertItem(p, j) \in O_{\mathcal{H}} \Rightarrow initInsertItem(p, j) \leq_{\mathcal{H}} insertItem(p, j)\ )\ )$.

(An insertItem operation should be initiated before it is completed.)

(e) $\forall p, p' \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($insertedItem(p', p, j) \in O_{\mathcal{H}} \Rightarrow initInsertItem(p, j) \leq_{\mathcal{H}} insertedItem(p', p, j)$)).

(An insertedItem operation should be initiated before it is completed.)

(f) $\forall p \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($deleteItem(p, j) \in O_{\mathcal{H}} \Rightarrow initDeleteItem(p, j) \leq_{\mathcal{H}} deleteItem(p, j)$)).

(A deleteItem operation should be initiated before it is completed.)

(g) $\forall p, p' \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($deletedItem(p', p, j) \in O_{\mathcal{H}} \Rightarrow initDeleteItem(p, j) \leq_{\mathcal{H}} deletedItem(p', p, j)$)).

(A deletedItem operation should be initiated before it is completed.)

(h) $\forall p \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall j \in \mathcal{T}$ ($getLocalItems_i(p, j[]) \in O_{\mathcal{H}} \Rightarrow initGetLocalItems_i(p) \leq_{\mathcal{H}} getLocalItems_i(p, j[])$)))

(A getLocalItems operation should be initiated before it is completed.)

(i) $\forall p \in \mathcal{P}$ ($dsInfoFromPredEvent(p) \in O_{\mathcal{H}} \Rightarrow \exists p' \in \mathcal{P}$ ($dsInfoForSuccEvent(p', p) \leq_{\mathcal{H}} dsInfoFromPredEvent(p)$))

(`DSINFOFROMPREDEVENT` *at peer* p *implies* `DSINFOFORSUCCEVENT` *occurred at the some peer p')*

(j) $\forall p \in \mathcal{P}$ ($o = rangeChangeEvent(p, r, b) \in O_{\mathcal{H}} \wedge o' = rangeChangeEvent(p, r', b') \in O_{\mathcal{H}} \wedge r' \neq \phi \wedge \nexists r''$ ($o'' = rangeChangeEvent(p, r'', b'') \in O_{\mathcal{H}} \wedge o <_{\mathcal{H}} o'' <_{\mathcal{H}} o'$) $\Rightarrow r.low = r'.low \vee r'.low = r'.high$)

(Successive range changes are such that one end of the range does not change.)

(k) $items_{\mathcal{H}}$ *is well-defined*

(l) Semantics of scanRange: $\forall i \in \mathcal{N}\ \forall lb, ub\ \forall p_1 \in \mathcal{P}\ o_e = doneScanRange_i(p_1, lb, ub) \in O_{\mathcal{H}} \Rightarrow$

i. $o_s = initScanRange_i(p_1, lb, ub) \leq_{\mathcal{H}} o_e$

ii. $\forall o \in scanOps(i)\ \forall p\ \forall r\ o = scanRange_i(p, p_1, r) \Rightarrow o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge r \subseteq range_{\mathcal{H}_o}(p)$

iii. $\forall o_l, o_m \in scanOps(i)\ o_l \neq o_m \wedge \forall p_l, p_m\ \forall r_l, r_m\ o_l = scanRange_i(p_l, p_1, r_l) \wedge o_m = scanRange_i(p_m, p_1, r_m) \Rightarrow \neg(r_l \bowtie r_m)$

iv. $[lb, ub] = \cup_{r \in rangeSet(i)}(r)$

(m) Semantics of getLocalItems: $\forall i \in \mathcal{N}$ ($\forall j \in \mathcal{T}$ ($\forall p \in P$ ($o = getLocalItems_i(p, j[]) \in O_{\mathcal{H}} \Rightarrow items_{\mathcal{H}_o}(p) = j[]$.

12.2 PEPPER Data Store History

We now present our implementation of the Data Store API methods. Later in this section, we show that PEPPER Data Store satisfies all the important correctness requirements of the API Data Store History.

12.2.1 Operations We now present our implementation of the *Data Store* and identify the different operations in our implementation.

Our implementation of the Data Store API methods and Ring event handlers and the other required methods is given in Algorithms 22 to 44.

The operations we are considering are listed as part of the pseudocode for the above algorithms. In addition, we have the fail operation, $fail(p)\ \forall p \in \mathcal{P}$. Given a set of peers $\mathcal{P}$, the set of allowed operations in any *PEPPER Data Store History* is denoted by $\mathcal{O}_{\mathcal{PH}}$.

Before we define *PEPPER Data Store History* corresponding to our implementation of the Data Store, we first define some notations.

Notation ($\mathcal{O}_{\mathcal{PH}}(p)$): $\mathcal{O}_{\mathcal{PH}}(p)$ is the subset of operations in $\mathcal{O}_{\mathcal{PH}}$ that occur on peer p.

12.2.2 Definition We now present the definition of *PEPPER Data Store History* $\mathcal{PH}$.

Definition 15 (PEPPER Data Store History) *Given set of peers $\mathcal{P}$ and a set of allowed operations $\mathcal{O}_{\mathcal{PH}}$ on these peers, $\mathcal{PH} = (\mathcal{O}_{\mathcal{PH}}, \leq_{\mathcal{PH}})$ is a* PEPPER Data Store History *iff*

1. *$\mathcal{PH}$ is a history*
2. *$O_{\mathcal{PH}} \subseteq \mathcal{O}_{\mathcal{PH}}$*
3. API restrictions
   - *(a)* $\exists p \in \mathcal{P}\ (\ initFirstPeer(p) \in O_{\mathcal{PH}} \wedge firstPeer(p) \in O_{\mathcal{PH}} \wedge (\forall p' \in \mathcal{P}\ initFirstPeer(p') \in O_{\mathcal{PH}} \Rightarrow p = p')\ )$.
     (There exists a unique peer p which starts the system.)
   - *(b)* $\forall p \in \mathcal{P}\ (\ op(p) \in \{initScanRange_i(p, lb, ub), initInsertItem(p, j), initDeleteItem(p, j), initGetLocalItems_i(p), initInsertLocalItem(p, j), initDeleteLocalItem(p, j), i, lb, ub \in \mathcal{N}, j \in \mathcal{T}\}\ ) \wedge op(p) \in O_{\mathcal{PH}} \Rightarrow dsInfoFromPredEvent(p) \leq_{\mathcal{PH}} op(p)\ )\ )$
     (All API operations other than $initFirstPeer$ can be initiated only after DSINFOFROMPREDEVENT*)*
   - *(c)* $\forall p \in \mathcal{P}\ (\ leaveEvent(p) \in O_{\mathcal{PH}} \Rightarrow\ (\ op(p) \in \{initFirstPeer(p), initScanRange_i(p, lb, ub), initInsertItem(p, j), initDeleteItem(p, j), initGetLocalItems_i(p), initInsertLocalItem(p, j), initDeleteLocalItem(p, j), i, lb, ub \in \mathcal{N}, j \in \mathcal{T}\}\ ) \wedge op(p) \in O_{\mathcal{PH}} \Rightarrow op(p) \leq_{\mathcal{PH}} leaveEvent(p)\ )$
     (No API operation can be initiated after LEAVEEVENT*)*
   - *(d)* $\forall p \in \mathcal{P}\ (\ fail(p) \in O_{\mathcal{PH}} \Rightarrow\ (\ op(p) \in \{initFirstPeer(p), initScanRange_i(p, lb, ub), initInsertItem(p, j), initDeleteItem(p, j), initGetLocalItems_i(p), initInsertLocalItem(p, j), initDeleteLocalItem(p, j), i, lb, ub \in \mathcal{N}, j \in \mathcal{T}\}\ ) \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{PH}} fail(p)\ )$
     (All operations on peer p are initiated before $fail(p)$)
   - *(e)* $\forall p, p' \in \mathcal{P}\ (\ dsInfoForSuccEvent(p, p') \in O_{\mathcal{PH}} \Rightarrow initDSInfoForSuccEvent(p, p') \leq_{\mathcal{PH}} dsInfoForSuccEvent(p, p')\ )$.
     (Any $dsInfoForSuccEvent(p, p')$ should be preceded by an $initDSInfoForSuccEvent(p, p')$ operation)
   - *(f)* $\forall p \in \mathcal{P}\ (\ \forall i\ (\ \forall r\ (\ \forall b\ (\ rangeChangeEvent_i(p, r, b) \in O_{\mathcal{PH}} \Rightarrow initRangeChangeEvent_i(p, r, b) \leq_{\mathcal{PH}} rangeChangeEvent_i(p, r, b)\ )\ )\ )\ )$.
     (Any $rangeChangeEvent_i(p, r, b)$ should be preceded by an $initRangeChangeEvent_i(p, r, b)$ operation.)
   - *(g)* $\forall p, p' \in \mathcal{P}\ (\ \forall j \in \mathcal{T}\ (\ initInsertItem(p, j) \in O_{\mathcal{PH}} \wedge initInsertItem(p, j) \in O_{\mathcal{PH}} \Rightarrow p = p'\ )\ )$
     (Insert of item $j \in \mathcal{T}$ is tried at most once.)
   - *(h)* $\forall p, p' \in \mathcal{P}\ (\ \forall j \in \mathcal{T}\ (\ initDeleteItem(p, j) \in O_{\mathcal{PH}} \wedge initDeleteItem(p, j) \in O_{\mathcal{PH}} \Rightarrow p = p'\ )\ )$
     (Delete of item $j \in \mathcal{T}$ is tried at most once.)
4. Happened Before Constraints

   The happened before constraints *can be inferred from the algorithms. In the interest of space, we are not writing out the* happened before constraints *as a table (as we did in the case of PEPPER Ring history). Note that all operations on peer p happen before $fail(p)$.*
5. Conflict Constraints

   The conflict constraints *can be inferred from the algorithms. In the interest of space, we are not writing out the* conflict constraints *as a table (as we did in the case of PEPPER Ring History).*

Algorithm 22 : $p.initFirstPeer()$

writeLock $range$, $state$, $items$
$state =$ `AVAILABLE`
$PinitFirstPeer1(p)$
$range = [a, a)$; // responsible for the full range
$items = \{\}$; // no items initially
$ring.initRing()$
$initRing(p)$
on receiving `INSERT` event,
$insert(p, p)$
// Wait for `INSERTED` event
on receiving `INSERTED` event and p is the first peer
$inserted(p)$
register message handlers & periodic procedures
releaseLock $state$
// raise the range change event
$initRangeChangeEvent(p, range, 0)$
$raiseEvent$(RANGECHANGEEVENT)
on receiving response,
$rangeChangeEvent(p, range, 0)$
releaseLock $range$, $items$
$PinitFirstPeer2(p)$
// raise an event to signal end of initFirstPeer
$firstPeer(p)$
$raiseEvent$(FIRSTPEER)

Algorithm 23 : $p.insertedEventHandler(splitMsg)$

On receiving the `INSERTED` event, p not the first peer
$inserted(p)$
get s_range, s_items, $upData$ from $splitMsg$
writeLock $state$, $range$, $items$
$state =$ `AVAILABLE`
$initDS1(p)$
$range = s\_range$; $items = s\_items$
register message handlers,periodic procedures
releaseLock $state$
// raise the range change event
$initRangeChangeEvent(p, range, 0)$
$raiseEvent$(RANGECHANGEEVENT)
on receiving response,
$rangeChangeEvent(p, range, 0)$
releaseLock $items$
$initDS2(p)$
releaseLock $range$
$initDS3(p)$
// raise the join ring event; include $upData$
$dsInfoFromPredEvent(p)$
$raiseEvent$(DSINFOFROMPREDEVENT, $upData$)
releaseLock $range$
$initDS4(p)$

Algorithm 24 : $p.initInsertItem(i.key, i.value)$

create $insertMsg$ with item i
$initInsertItem1(p, i)$
$returnCode$ = $p.dsInsertMsgHandler(insertMsg)$
// Raise event indicating completion
$insertItem(p, i)$
$raiseEvent$(INSERTITEM, $returnCode$)

Algorithm 25 : $p.initDeleteItem(i.key, i.value)$

create $deleteMsg$ with item i
$initDeleteItem1(p, i)$
$returnCode$ = $p.dsDeleteMsgHandler(deleteMsg)$
// Raise event indicating completion
$deleteItem(p, i)$
$raiseEvent$(DELETEITEM, $returnCode$)

Algorithm 26 : $p.ringRecvSendToSuccEventHandler(p', msg)$

find right handler from the header of msg
$recvSendToSucc_i(p, p')$
$retMsg = handler.handleEvent(msg)$
return $retMsg$

Algorithm 27 : $p.dsInsertMsgHandler(insertMsg)$

0: get item i from $insertMsg$ received from peer p'
$insertItem1(p, i)$
1:
2: readLock $dsState$, $range$
$insertItem2(p, i)$
3: **if** $dsState \neq$ `AVAILABLE` **then**
4:
5: releaseLock $range$, $dsState$ //Not ready for insert
$insertItemAbort1(p, i)$
6: return DSInsertRetMsg(`FAILURE`);
7: **end if**
8: // Check if item needs to be inserted at p
$insertItem3(p, i)$
9: **if** $\neg([i.key, i.key] \bowtie range)$ **then**
10:
11: // Forward message to successor
$initSendToSucc_i(p), sendToSucc_i(p)$
12: $retCode = ring.initSendToSucc(insertMsg)$
13: releaseLock $dsState$, $range$
$insertItemReturn(p, i)$
14: return $retCode$
15: **end if**
16: // Reached the right peer; Insert item into local store
17: writeLock $items$
$insertItem4(p, i)$
18: insert i into $items$
19: $response = raiseEvent$(`ITEMINSERTED`, i)
$insertedItem(p, p', i)$
20:
21: releaseLock $items$, $range$, $dsState$
22: $retMsg$ = DSInsertRetMsg($response.retCode$);
$insertItem5(p, i)$
23: return $retMsg$;

Algorithm 28 : $p.dsDeleteMsgHandler(deleteMsg)$

0: get item i from $deleteMsg$ received from peer p'
$deleteItem1(p, i)$
1:
2: readLock $dsState$, $range$
$deleteItem2(p, i)$
3: **if** $dsState \neq$ `AVAILABLE` **then**
4:
5: releaseLock $range$, $dsState$ //Not ready for delete
$deleteItemAbort1(p, i)$
6: return DSDeleteRetMsg(`FAILURE`);
7: **end if**
8: // Check if item needs to be deleted at p
$deleteItem3(p, i)$
9: **if** $\neg([i.key, i.key] \bowtie range)$ **then**
10:
11: // Forward message to successor
$initSendToSucc_i(p), sendToSucc_i(p)$
12: $retCode = ring.initSendToSucc(deleteMsg)$
13: releaseLock $dsState$, $range$
$deleteItemReturn(p, i)$
14: return $retCode$
15: **end if**
16: // Reached the right peer; Delete item into local store
17: writeLock $items$
$deleteItem4(p, i)$
18: delete i into $items$
19: $response = raiseEvent$(`ITEMDELETED`, i)
$deletedItem(p, p', i)$
20:
21: releaseLock $items$, $range$, $dsState$
22: $retMsg$ = DSDeleteRetMsg($response.retCode$);
$deleteItem5(p, i)$
23: return $retMsg$;

Algorithm 29 : $p.initScanRange_i(lb, ub, hId, param)$

```
⌈ 0: readLock range
initScanRange1_i(p)
| 1: if lb ∉ p.range then
⌊ 2:
⌈ 3:     // Abort scanRange
initScanRangeAbort1_i(p)
⌊ 4:     releaseLock range
  5: else
⌈ 6:     // p is the first peer in scan range
initScanRange2_i(p)
⌊ 7:     p.processHandler_i1(lb, ub, hId, param)
⌈ 8:     // Wait for an ack from the last peer
initScanRange3_i(p)
| 9:     if ack is received in TIMEOUT period then
⌊ 10:
⌈ 11:        // Raise an event to signal end of scanRange
doneScanRange_i(p)
⌊ 12:        raiseEvent(DONESCANRANGE)
  13:    else
⌈ 14:        // Abort scanRange
initScanRangeAbort2_i(p)
⌊ 15:
  16:    end if
  17: end if
```

Algorithm 30 : $p.processHandler_{ik}(lb, ub, hId, param)$

```
⌈ 0: readLock dsState
processHandler1_i(p)
| 1: if dsState ≠ AVAILABLE then
⌊ 2:
⌈ 3:     releaseLock dsState //Not ready for scanRange
processHandlerAbort(p, i)
⌊ 4:     return;
  5: end if
⌈ 6: // Invoke appropriate handler
processHandler2_ik(p)
| 7: get handler with id hId
⌊ 8: r_k = [lb, ub] ∩ p.range
⌈ 9: newParam = handler_i.handle(r_k, param)
processHandler3_ik(p)
⌊ 10:
⌈ 11: // Raise an event to signal scanRange
scanRange_ik(p, r_k)
⌊ 12: raiseEvent(SCANRANGE)
⌈ 13: // Forward to successor if required
processHandler4_i(p)
| 14: if ub ∉ p.range then
⌊ 15:
⌈ 16:    p_succ = ring.initGetSuccessor()
initGetSucc_i(p), getSucc_i(p)
⌊ 17:
⌈ 18:    p_succ.processScan_i(k+1)(lb, ub, hId, newParam)
processHandler5_i(p)
⌊ 19:
  20: else
⌈ 21:    get p′, the peer which initiated this scanRange
processHandler6_i(p, p′)
⌊ 22:    send ack to p′ indicating end of scanRange
  23: end if
⌈ 24: releaseLock state, range
processHandler7_i(p)
⌊ 25:
```

Algorithm 31 : $p.processScan_{ik}(lb, ub, hId, param)$

```
⌈ 0: readLock range
processScan1_i(p)
| 1: invoke p.processHandler_ik(lb, ub, hId, param)
     asynchronously
⌊ 2: return
```

Algorithm 32 : $p.initiateSplit_i()$

writeLock $dsState$
$initiateSplit1_i(p)$
if $dsState \neq$ `AVAILABLE` **then**
releaseLock $dsState$
$initiateSplitAbort_i(p)$
return; //Not available for split
else
$dsState =$ `SPLITTING`
$initiateSplit2_i(p)$
releaseLock $dsState$
end if
$p' = findFreePeer()$ //Find a free peer p'
$initiateSplit3_i(p, p')$
// If found, insert the free peer into the ring
$initiateSplit4_i(p, p')$
if found free peer **then**
$ring.initInsert(p')$;
$initInsert(p, p')$
end if
writeLock $dsState$
$initiateSplit5_i(p)$
$dsState =$ `AVAILABLE`
releaseLock $dsState$

Algorithm 33 : $p.insertEventHandler(p')$

on receiving `INSERTEVENT`
$initInsertEvent(p, p')$

writeLock $range$, $items$
$insertHandler1(p, p')$
split $range$, $items$
// Raise the range change event
$initRangeChangeEvent(p, range, 0)$
$raiseEvent$(`RANGECHANGEEVENT`)
on receiving response,
$rangeChangeEvent(p, range, 0)$

releaseLock $items$
$insertHandler2(p, p')$

downgradeLock $range$
$insertHandler3(p, p')$

// Raise the dsInfoForSuccEvent
$initDSInfoForSuccEvent(p, p')$
$raiseEvent$(`DSINFOFORSUCCEVENT`)
on receiving $upData$ as response,
$dsInfoForSuccEvent(p, p')$

construct $joinmsg$ to be sent to the potential successor; Include $upData$.
releaseLock $range$
$insert(p, p')$
return $joinmsg$

Algorithm 34 : $p.merge_i()$

0: writeLock $dsState$
$merge1_i(p)$
1: **if** $dsState \neq$ `AVAILABLE` **then**
2:
3: releaseLock $dsState$
$mergeAbort_i(p)$
4: return; // Not available for merge
5: **else**
6: $dsState =$ `MERGING`
$merge2_i(p)$
7: releaseLock $dsState$
8: **end if**
9: writeLock $range$
$merge3_i(p)$
10:
11: p_s = $ring.initGetSuccessor()$
$initGetSucc_i(p), getSucc_i(p)$
12:
13: readLock $items$
$merge4a_i(p)$
14: $numItems = items.size()$
15: releaseLock $items$
16: // Send a DSInitiateMerge message to the successor
17: construct $mergeMsg$; include $numItems$
$merge4_i(p)$
18: $retMsg = p_s.dsInitMergeMsgHandler(mergeMsg)$
19: // If a retMsg is received
$merge5_i(p)$
20: **if** $retMsg \neq$ `NULL` **then**
21:
22: writeLock $items$
$merge6_i(p)$
23: $p.range.high = p_s.range.low$; update items
24: // Raise the range change event
$initRangeChangeEvent(p, range, 0)$
25: $raiseEvent$(`RANGECHANGEEVENT`)
26: on receiving response,
$rangeChangeEvent(p, range, 0)$
27:
28: releaseLock $items$
$merge7_i(p)$
29:
30: **end if**
31: releaseLock $range$
$merge8_i(p)$
32:
33: writeLock $dsState$
$merge9_i(p)$
34: $dsState =$ `AVAILABLE`
35: releaseLock $dsState$

Algorithm 35 : $p.dsInitiateMergeMsgHandler(msg)$

0: get $origAddr$ and $origNbItems$ from rcvd msg
$mergeHandler1_i(p)$
1: writeLock $dsState$
2: **if** $dsState \neq$ `AVAILABLE` **then**
3:
4: releaseLock $dsState$
$mergeHandlerAbort_i(p)$
5: return `NULL` // Merge abort
6: **else**
7: $dsState =$ `MERGING`
$mergeHandler2_i(p)$
8: releaseLock $dsState$
9: **end if**
10: readLock $items$
$mergeHandler3_i(p)$
11: $numItemsIHave = items.size()$
12: releaseLock $items$
13: // Check if redistribution possible
$mergeHandler4_i(p)$
14: **if** can redistribute **then**
15: serializedRetMsg = recvRedistribute(mergeMsg)
16: **else**
17: // Merge
$mergeHandler5_i(p)$
18: serializedRetMsg = recvMerge(mergeMsg)
19: **end if**
20: return serializedRetMsg;
$mergeHandler7_i(p)$
21:

Algorithm 36 : $p.recvRedistribute()$

0: writeLock $range$, $items$
1: update $range$ and $items$
$recvRedistribute1_i(p)$
2: construct $retMsg$
3: // raise the range change event
$initRangeChangeEvent(p, range, 0)$
4: $raiseEvent$(`RANGECHANGEEVENT`)
5: on receiving response,
$rangeChangeEvent(p, range, 0)$
6:
7: releaseLock $range$, $items$
$recvRedistribute2_i(p)$
8:
9: writeLock $dsState$
$recvRedistribute3_i(p)$
10: $dsState =$ `AVAILABLE`
11: releaseLock $dsState$

Algorithm 37 : $p.recvMerge()$

$ring.initLeave()$;
$initLeave(p)$
:
wait for the LEAVE event
$leave(p)$, $leaveEvent(p)$
if received within TIMEOUT period **then**
// Raise leave event
$raiseEvent$(LEAVEEVENT);
construct $retMsg$
$recvMerge1(p)$
unregister msg handlers, periodic procedures
writeLock $dsState$, $range$, $items$
$recvMerge2(p)$
$state =$ FREE, $range = \phi$, $items = \{\}$
releaseLock $dsState$, $items$
// raise the range change event
$initRangeChangeEvent(p, range, 0)$
$raiseEvent$(RANGECHANGEEVENT)
on receiving response,
$rangeChangeEvent(p, range, 0)$
:
releaseLock $range$
$recvMerge3(p)$
:
return $retMsg$
$recvMerge4(p)$
:
else
// Abort merge
$recvMergeAbort_i(p)$
writeLock $dsState$
$dsState =$ AVAILABLE
releaseLock $dsState$
return NULL
end if

Algorithm 38 : $p.ringNewSuccEventHandler()$

// Raise event to indicate a new successor
$newSuccEvent(p)$
$raiseEvent$(NEWSUCCEVENT)

Algorithm 39 : DataStoreBalance

readLock $items$
$dsBalance1_i(p)$
$numItems = items.size()$
releaseLock $items$
// Too few items... merge
$dsBalance2_i(p)$
if $numItems <$ STOREFACTOR **then**
$merge()$;
// Too many items... split
$dsBalance3_i(p)$
else if $numItems > 2 *$ STOREFACTOR **then**
$initiateSplit()$
end if

Algorithm 40 : $p.infoForSuccEventHandler()$

readLock $range$
// Send high-end of range to successor
$infoForSucc1_i(p)$
str = serialized $range.getHighEndValue()$
releaseLock $range$
return str
$infoForSucc2_i(p)$
:

Algorithm 41 : $p.infoFromPredEventHandler(str)$

get $value$ from serialized string str
writeLock $range$
$infoFromPred1_i(p)$
// Set low-end of range
$range.setLowEndValue(value)$
// Raise the range change event
$initRangeChangeEvent(p, range, 1)$
$raiseEvent$(RANGECHANGEEVENT)
on receiving response,
$rangeChangeEvent(p, range, 1)$
:
releaseLock $range$
$infoFromPred2_i(p)$
:

Algorithm 42 : $p.initGetLocalItems()$

// Return local items from data store
readLock $items$
$retItems = items$
$getLocalItems(p, items)$
releaseLock $items$
return $retItems$

Algorithm 43 : $p.initInsertLocalItem(j)$

// Insert j into p's data store
writeLock $items$
$initInsertLocalItem1(p, j)$
insert j into $items$
$raiseEvent$(LOCALITEMINSERTED, j)
$insertLocalItem(p, j)$
:
releaseLock $items$
$initInsertLocalItem2(p, j)$
:

Algorithm 44 : $p.initDeleteLocalItem(j)$

0: // Delete j into p's data store
1: writeLock $items$

$initDeleteLocalItem1(p, j)$

2: delete j into $items$
3: $raiseEvent$(`LOCALITEMDELETED`, j)

$deleteLocalItem(p, j)$

4:
5: releaseLock $items$

$initDeleteLocalItem2(p, j)$

6:

12.2.3 Useful Result

Lemma 9 *Any PEPPER Data Store History $\mathcal{H}$ uses the ring API appropriately i.e $\Pi_{\mathcal{O}(\mathcal{P})}(\mathcal{H})$ satisfies the API ring history requirements.*

Proof: We need to show that all the API ring history restrictions are respected in the PEPPER Data Store implementation.

1. (There exists a unique peer p which starts off the ring.)

   From API restriction 3(a) of *PEPPER Data Store History*, $\exists p \in \mathcal{P}$ ($initFirstPeer(p) \in O_{\mathcal{H}} \wedge firstPeer(p) \in O_{\mathcal{H}} \wedge (\forall p' \in \mathcal{P}\ initFirstPeer(p') \in O_{\mathcal{H}} \Rightarrow p = p')$).

   From Algorithm 22, $firstPeer(p) \in O_{\mathcal{PH}} \Rightarrow initRing(p) \leq_{\mathcal{PH}} firstPeer(p) \wedge insert(p,p) \leq_{\mathcal{PH}} firstPeer(p)$. Therefore, $\exists p \in \mathcal{P}$ ($initRing(p) \in O_{\mathcal{PH}} \wedge insert(p,p) \in O_{\mathcal{PH}}$).

   From Algorithm 22, $initRing(p') \in O_{\mathcal{PH}} \Rightarrow initFirstPeer(p') \leq_{\mathcal{PH}} initRing(p')$. Using API restriction 3(a), we conclude that $p = p'$. Therefore, $\forall p' \in \mathcal{P}\ initRing(p') \in O_{\mathcal{PH}} \Rightarrow p = p'$

   Hence, $\exists p \in \mathcal{P}$ ($initRing(p) \in O_{\mathcal{PH}} \wedge insert(p,p) \in O_{\mathcal{PH}} \wedge (\forall p' \in \mathcal{P}\ initRing(p') \in O_{\mathcal{PH}} \Rightarrow p = p')$).

2. (Insert of peer p' is tried at most once. This is not a necessary but a convenient API restriction.)

   From Algorithm 32, $initInsert(p'',p') \Rightarrow \exists i \in \mathcal{N} initiateSplit3_i(p'',p') \leq_{\mathcal{PH}} initInsert(p'',p')$.

   We assume that we find a unique free peer each time the $findFreePeer$ call is invoked. Therefore, $\forall p,p',p'' \in \mathcal{P}$ ($p \neq p' \wedge p'' \neq p' \wedge initInsert(p,p') \in O_{\mathcal{PH}} \wedge initInsert(p'',p') \in O_{\mathcal{PH}} \Rightarrow p'' = p$)

   Moreover, $initiateSplit3_i(p'',p') \in O_{\mathcal{PH}} \Rightarrow initFirstPeer(p') \notin O_{\mathcal{PH}}$. (We assume that $findFreePeer$ returns only peers which are not potential first peers).

   From Algorithm 22, $initFirstPeer(p') \notin O_{\mathcal{PH}} \Rightarrow initRing(p') \notin O_{\mathcal{PH}}$

   Therefore, $\forall p,p' \in \mathcal{P}, p \neq p'$, ($initInsert(p,p') \in O_{\mathcal{H}} \Rightarrow (initRing(p') \notin O_{\mathcal{H}} \wedge (\forall p'' \in \mathcal{P}\ initInsert(p'',p') \Rightarrow p'' = p)))$.

3. (All operations on peer p except $initRing(p)$ are initiated after $inserted(p)$.)

   Let $o = inserted(p) \in O_{\mathcal{PH}}$. Note that $p.state_{\mathcal{PH}_o} = \texttt{FREE}$.

   (a) Ring API operation $initLeave(p)$ occurs only in Algorithm 37 invoked from Algorithm 35.

   (b) Ring API operation $initInsert(p,p')$ occurs only in Algorithm 32.

   (c) Ring API operation $initSendToSucc_i(p,p')$ occurs only in Algorithms 27 and 28.

   (d) Ring API operation $initGetSucc_i(p)$ occurs only in Algorithms 30 and 34.

   All handlers are registered only after $inserted(p)$ in operation $initDS_1(p)$.

   $initLeave(p)$ occurs only in Algorithm 37 invoked from Algorithm 35. This message handler is registered only after $inserted(p)$. Therefore, $inserted(p) \leq_{\mathcal{PH}} initLeave(p)$

   $initInsert(p,p')$ occurs only in Algorithm 32. Algorithm 32 is invoked from Algorithm 39. This periodic procedure is registered only after $inserted(p)$. Therefore, $inserted(p) \leq_{\mathcal{PH}} initInsert(p,p')$.

   $initSendToSucc_i(p,p')$ occurs only in Algorithms 27 and 28. These message handlers are registered only after $inserted(p)$. These message handlers can be locally invoked from Algorithms 24 and 25.

   From API restriction 3(b), $dsInfoFromPredEvent(p) \leq_{\mathcal{PH}} initInsertItem(p,i)$. From Algorithm 23, $inserted(p) \leq_{\mathcal{PH}} dsInfoFromPredEvent(p)$. Therefore, $inserted(p) \leq_{\mathcal{PH}} initInsertItem(p,i)$. Similarly, $inserted(p) \leq_{\mathcal{PH}} initDeleteItem(p,i)$. Also, from Algorithms 24 and 25, $initInsertItem(p,i) \leq_{\mathcal{H}} initSendToSucc_i(p,p') \wedge initDeleteItem(p,i) \leq_{\mathcal{H}} initSendToSucc_i(p,p')$.

   Therefore, $inserted(p) \leq_{\mathcal{PH}} initSendToSucc_i(p,p')$.

   $initGetSucc_i(p)$ occurs only in Algorithms 30 and 34.

   $processHandler$ is registered only after $inserted(p)$. Algorithm 30 is invoked locally from Algorithm 29. From API restriction 3(b), $infoFromPredEvent(p) \leq_{\mathcal{PH}} initScanRange(p,lb,ub)$. From Algorithm 23, $inserted(p) \leq_{\mathcal{PH}} infoFromPredEvent(p)$. Therefore, $inserted(p) \leq_{\mathcal{PH}} initScanRange(p,lb,ub)$.

   Algorithm 34 is invoked from Algorithm 39. This periodic procedure is registered only after $inserted(p)$.

   Therefore, $inserted(p) \leq_{\mathcal{PH}} initGetSucc_i(p)$.

   Hence, $\forall p \in \mathcal{P}$ ($\forall op(p) \in \{initLeave(p), initGetSucc_i(p)\}$ ($op(p) \in O_{\mathcal{H}} \Rightarrow inserted(p) \in O_{\mathcal{H}} \wedge inserted(p) \leq_{\mathcal{H}} op(p)$) $\wedge$ ($\forall p' \in \mathcal{P}, \forall op(p,p') \in \{initInsert(p,p'), initSendToSucc_i(p,p'), \forall i \in \mathcal{N}\}$ ($op(p,p') \in O_{\mathcal{H}} \Rightarrow inserted(p) \in O_{\mathcal{H}} \wedge inserted(p) \leq_{\mathcal{H}} op(p,p')$)))

4. (All operations on peer p are initiated before $fail(p)$.)

Follows from *happened before relationships*.

$\forall p \in \mathcal{P}$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow$ ($op(p) \in \{initRing(p),$
$initLeave(p), initGetSucc_i(p),$
$initSendToSucc_i(p), \forall i \in \mathcal{N}\} \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{H}} fail(p)$) $\wedge$ ($\forall p' \in \mathcal{P}$ ($op(p,p') \in \{initInsert(p,p')\} \wedge op(p,p') \in O_{\mathcal{H}} \Rightarrow op(p,p') \leq_{\mathcal{H}} fail(p)$)))

5. (All operations on peer p are initiated before $leave(p)$.)

   (a) Ring API operation $initRing(p)$ occurs only in Algorithm 22.

   (b) Ring API operation $initLeave(p)$ occurs only in Algorithm 37 invoked from Algorithm 35.

   (c) Ring API operation $initInsert(p,p')$ occurs only in Algorithm 32.

   (d) Ring API operation $initSendToSucc_i(p,p')$ occurs only in Algorithms 27 and 28.

   (e) Ring API operation $initGetSucc_i(p)$ occurs only in Algorithms 30 and 34.

   We assume that we find a unique free peer each time the $findFreePeer$ call is invoked. Therefore, once a peer leaves the ring, it is never inserted into the ring again.

   From Algorithm 22, $initRing(p) \in O_{\mathcal{PH}} \wedge inserted(p) \in O_{\mathcal{H}} \Rightarrow initRing(p) \leq_{\mathcal{PH}} inserted(p)$.

   As shown above, $inserted(p) \leq_{\mathcal{H}} initLeave(p)$. From Ring API semantic requirements, $initLeave(p) \leq_{\mathcal{H}} leave(p)$. Therefore, $initRing(p) \leq_{\mathcal{PH}} leave(p)$.

   In Algorithm 37, $dsState$ is set to `MERGING` before $leave(p)$ and `FREE` after. Therefore, $initInsert(p,p')$ and $initSendToSucc_i(p,p')$ which need $dsState$ to be `AVAILABLE` can happen only before merge. Hence, $initInsert(p,p') \leq_{\mathcal{PH}} leave(p)$ and $initSendToSucc_i(p,p') \leq_{\mathcal{PH}} leave(p)$.

   $initGetSucc_i(p)$ occurs only in Algorithms 30 and 34.

   In Algorithm 34, $dsState$ needs to be `AVAILABLE` initially. In Algorithm 30, $dsState$ needs to be `AVAILABLE`. In Algorithm 37, $dsState$ is set to `MERGING` before $leave(p)$ and `FREE` after. and hence $initGetSucc_i(p)$ can only happen before $leave(p)$. Therefore, $initGetSucc_i(p) \leq_{\mathcal{PH}} leave(p)$.

   Hence, $\forall p \in \mathcal{P}$ ($leave(p) \in O_{\mathcal{H}} \Rightarrow$ ($op(p) \in \{initRing(p),$
   $initLeave(p), initGetSucc_i(p),$
   $initSendToSucc_i(p) \forall i \in \mathcal{N}\} \wedge op(p) \in O_{\mathcal{H}} \Rightarrow op(p) \leq_{\mathcal{H}} leave(p)$) $\wedge$ ($\forall p' \in \mathcal{P}$ ($op(p,p') \in \{initInsert(p,p')\} \wedge op(p,p') \in O_{\mathcal{H}} \Rightarrow op(p,p') \leq_{\mathcal{H}} leave(p)$)))

Table 5: Table of operations at peer p modifying $p.range$

$PinitFirstPeer1(p)$
$initDS$
$insertHandler1_i(p)$
$merge6_i(p)$
$recvRedistribute1_i(p)$
$recvMerge2(p)$
$infoFromPred1_i(p)$

Table 6: Table of operations at peer p modifying $p.items$

$PinitFirstPeer1(p)$
$initDS1p)$
$insertItem4(p,i)$
$merge6_i(p)$
$recvRedistribute1_i(p)$
$recvMerge2(p)$
$initInsertLocalItem1(p,j)$
$initDeleteLocalItem1(p,j)$

6. (API fail operation on peer p cannot occur after $leave(p)$)

   Follows from API restriction 3(c). We ignore all fail operations after leave operations.

   $\forall p \in \mathcal{P}$ ($leave(p) \in O_{\mathcal{H}} \Rightarrow$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow fail(p) \leq_{\mathcal{H}} leave(p)$)).

■

12.2.4 $scanRange$ correctness

Theorem 2 *Any PEPPER Data Store History $\mathcal{H}$, satisfies scanRange correctness.*

Proof: <Proof of this theorem uses claim 13>

We need to show the following:

$\forall i \in \mathcal{N}\ \forall lb, ub\ \forall p_1 \in \mathcal{P}\ o_e = doneScanRange_i(p_1, lb, ub) \in O_{\mathcal{H}} \Rightarrow$

1. $o_s = initScanRange_i(p_1, lb, ub) \leq_{\mathcal{H}} o_e$
2. $\forall o \in scanOps(i)\ \forall p\ \forall r\ o = scanRange_i(p, p_1, r) \Rightarrow o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge r \subseteq range_{\mathcal{H}_o}(p)$
3. $\forall o_l, o_m \in scanOps(i)\ o_l \neq o_m \wedge \forall p_l, p_m\ \forall r_l, r_m\ o_l = scanRange_i(p_l, p_1, r_l) \wedge o_m = scanRange_i(p_m, p_1, r_m) \Rightarrow \neg(o_l \bowtie o_m)$
4. $[lb, ub] = \cup_{r \in rangeSet(i)}(r)$

From Algorithms 30 and 29, $doneScanRange_i(p_1, lb, ub) \in O_{\mathcal{H}} \Rightarrow \exists p_n\ (\ processHandler5_i(p_n, p_1) \leq_{\mathcal{H}} doneScanRange_i(p_1, lb, ub)\ )$.

Let $scanOps(i) = U_{k=1}^{k=n}(o_k = scanRange_{ik}(p_k, p_1, r_k))$.

From Algorithms 30, $processHandler5_i(p_n, p_1) \in O_{\mathcal{H}} \Rightarrow scanRange_{in}(p_n, p_1, r_n) \leq_{\mathcal{H}} processHandler5_i(p_n, p_1)$.

From Algorithms 30 and 31, $\forall k\ (\ 1 < k \leq n \wedge scanRange_{ik}(p_k, p_1, r_k) \in O_{\mathcal{H}} \Rightarrow processScan1_i(p_k) \leq_{\mathcal{H}} scanRange_{ik}(p_k, p_1, r_k)\ )$

From Algorithms 30, $\forall k\ (\ 1 < k \leq n \wedge processScan1_i(p_k) \in O_{\mathcal{H}} \Rightarrow scanRange_{i(k-1)}(p_{k-1}, p_1, r_{k-1}) \leq_{\mathcal{H}} processScan1_i(p_k)$.

From Algorithms 30 and 29, $scanRange_{i1}(p_1, p_1, r_1) \in O_{\mathcal{H}} \Rightarrow initScanRange2_i(p_1) \leq_{\mathcal{H}} scanRange_{i1}(p_1, p_1, r_1)$.

From Algorithm 29, $initScanRange2_i(p_1) \in O_{\mathcal{H}} \Rightarrow initScanRange_i(p_1, lb, ub) \leq_{\mathcal{H}} initScanRange2_i(p_1)$.

Therefore, $o_s \leq_{\mathcal{H}} o_e$ and $\forall k\ (\ 1 \leq k \leq n \Rightarrow o_s \leq_{\mathcal{H}} o_k \leq_{\mathcal{H}} o_e\ )$.

We now show that:

1. $\forall k\ (\ 1 \leq k \leq n \Rightarrow r_k \subseteq range_{\mathcal{H}}(p_k)\ )$
2. $r_1, r_2, \cdots, r_n$ is a partition of $[lb, ub]$

$initScanRange_i$ starts at peer $p = p_1$ with $lb \in p.range$. Therefore $r_1.low = lb$. Also, note that $scanRange$ forwarding ends at peer p_n such that $ub \in p_n.range$ (see Algorithm 30). Therefore, $r_n.high = ub$. Using the claim 13 with peers p_{k-1} and p_k (i=2 to n), we conclude $r_{k-1}.high = r_k.low$ and $r_k \subseteq range_{\mathcal{H}}(p_k)\ )$

Note that $ub \notin r_k$ for any $k \neq n$ and hence $r_k, k \neq n$, cannot overlap with any of $r_j, 1 \leq j < k$. Therefore, $\forall j\ 1 \leq j \leq n\ r_j$ is a partition of $[lb, ub]$.

This proves all the four conditions required for scanRange correctness.

■

Claim 13 *Given* PEPPER Data Store History $\mathcal{H}$, $\forall p, p_1, p_2 \in \mathcal{P}\ (\ \forall i, k \in \mathcal{N}\ (\ o_1 = scanRange_{ik}(p_1, p, r_1) \in O_{\mathcal{H}} \wedge o_2 = scanRange_{i(k+1)}(p_2, p, r_2) \in O_{\mathcal{H}} \Rightarrow r_1.high = r_2.low \wedge r_1 \subseteq range_{\mathcal{H}_{o_1}}(p_1) \wedge r_2 \subseteq range_{\mathcal{H}_{o_2}}(p_2)\ )\ )$

Proof: <Proof of this claim uses claim 14>

From implementation of $scanRange$ (see Algorithms 30 and 31), $o_1 = scanRange_{ik}(p_1, p, r_1) \in O_{\mathcal{H}} \wedge o_2 = scanRange_{i(k+1)}(p_2, p, r_2) \in O_{\mathcal{H}} \Rightarrow o_1 \leq_{\mathcal{H}} getSucc_i(p_1, p_2) \leq_{\mathcal{H}} o_2$.

From API Ring History semantic requirement $m(ii)$, $o = getSucc_i(p_1, p_2) \in O_{\mathcal{H}} \Rightarrow \exists\ j \in \mathcal{N}\ (o' = infoFromPredEvent_j(p_2, p_1) \in O_{\mathcal{H}} \wedge (\exists osucc \in O_{\mathcal{H}}\ (initGetSucc_i(p) \leq_{\mathcal{H}} osucc \leq_{\mathcal{H}} o \wedge o' \leq_{\mathcal{H}} osucc \wedge (\forall o''' \in O_{\mathcal{H}} (o' \leq_{\mathcal{H}} o''' \leq_{\mathcal{H}} osucc \Rightarrow p_2 = succ_{\mathcal{H}_{o'''}}(p_1)\ ))))\ )$. (*)

Let us consider the operations that could change the successor between $osucc$ and o. Since $succ_{\mathcal{H}_{osucc}}(p_1) = p_2$ and p_2 is live at o, the only change in successor of p_1 could come if a new successor would be inserted. However, the operation $insertHandler1$ conflicts with scan range (because of conflicting locks on $p.range$), so a new successor cannot be inserted between $osucc$ and o. (**)

From API Ring History semantic requirement n, $o' = infoFromPredEvent_j(p_2, p_1) \in O_{\mathcal{H}} \Rightarrow o'' = infoForSuccEvent_j(p_1) \leq_{\mathcal{H}} infoFromPredEvent_j(p_2, p_1) \wedge succ_{\mathcal{H}_{o_3}}(p_1) = p_2, \forall o'' \leq_{\mathcal{H}} o_3 \leq_{\mathcal{H}} o'$. (***)

From (*), (**) and (***) we have that $p.succ_{\mathcal{H}_{op}}(p_1) = p_2, \forall o'' \leq_{\mathcal{H}} op \leq_{\mathcal{H}} o$. (****)

From Algorithm 41, $p_1.range_{\mathcal{H}_{o''}}.high = p_2.range_{\mathcal{H}_{o'}}.low$.

Now let us consider the operations between o'' and o at peer p_1 which possibly modify $p_1.range$. Note that peer p_1 cannot leave the ring before o. Other than initialization and reset to NULL after leave, the range of a peer is modified by the following four operations:

1. $insertHandler1_i(p_1)$ (on a split, at the splitting peer): This operation only changes the high end of the range.
2. $merge6_i(p_1)$ (on the peer which has initiated a merge/redistribute): This operation only changes the high end of the range.
3. $recvRedistribute1_i(p_1)$ (on the peer which is redistributing based on the predecessor's request): This operation only changes the low end of the range.
4. $infoFromPred1_i(p_1)$ (on receiving the infoFromPredEvent): This operation only changes the low end of the range.

We now see which of these operations could have changed $p_1.range$ between o'' and o.

- Since $succ_{\mathcal{H}_{op}}(p_1) = p_2, \forall o'' \leq_{\mathcal{H}} op \leq_{\mathcal{H}} o$, no split at p_1 could have completed between o'' and o (a split completion means that a new successor is introduced). This rules out possibility 1 above.

- A merge/redistribute initiated by p_1 could have completed at p_1 between o'' and o. This cannot be a merge because $succ_{\mathcal{H}_{op}}(p_1) = p_2, \forall o'' \leq_{\mathcal{H}} op \leq_{\mathcal{H}} o$. Note that there is a read lock on range at o. Therefore, the redistribute operation should have completed before o.

  Redistributes and infoFromPredEvents executed in parallel could lead to inconsistent ranges. Suppose a $infoForSuccEvent_j(p_1)$ happened before the redistribute is initiated and the $infoFromPredEvent_j(p_2, p_1)$ happened after recvRedistribute at p_2. $p2.range.low$ modified by redistribute is set to the old stale value sent by $infoForSuccEvent$.

  We now show that o' is the only $infoFromPredEvent$ at p_2 between o'' and o (1). Moreover, we also show that any redistribute which completes at p_1 between o'' and o should be initiated after o' (2). We thus ensure that there are no redistributes and infoFromPredEvents overlapping at p_2 between o'' and o. Hence, from operation $merge6_i(p)$, we conclude that redistribute completion respects the invariant that $p_1.range.high = p_2.range.low$.

  (1) can be proved as follows: Since $succ_{\mathcal{H}_o}(p_1) = p_2$ and using API Ring History semantic requirement $m(iii)$, $\forall o'''$ ($o' \leq_{\mathcal{H}} o''' <_{\mathcal{H}} o \Rightarrow p_2 = succ_{\mathcal{H}_{o'''}}(p_1)$). Suppose if possible, $\exists p \in \mathcal{P}$ ($o_p = infoFromPredEvent(p_2, p) \wedge o' \leq_{\mathcal{H}} o_p <_{\mathcal{H}} o$). Since o' is the last info from predecessor event at p_2 from p_1, we conclude that $p \neq p_1$. From API Ring History semantic requirement $n(ii)$, we infer that $p_2 = succ_{\mathcal{H}_{o_p}}(p)$. Therefore, $p_2 = succ_{\mathcal{H}_{o_p}}(p) = succ_{\mathcal{H}_{o_p}}(p_1)$, contradicting the fact that $succ_{\mathcal{H}_{o_p}}$ is a bijection.

  Hence, $\nexists p \in \mathcal{P}$ ($o_p = infoFromPredEvent(p_2, p) \wedge o' \leq_{\mathcal{H}} o_p <_{\mathcal{H}} o$). Therefore, o' is the only $infoFromPredEvent$ at p_2 between o'' and o.

  (2) can be proved as follows:

  Suppose if possible, there is a redistribute initiated by p_1 after o'' and before o_1. Then, considering $getSucc_i(p_1, p_2)$ in Algorithm 34, we have $o'' <_{\mathcal{H}} getSucc_i(p_1, p_2)$.

  From API Ring History semantic requirement $m(ii)$, $getSucc_i(p_1, p_2)$ in Algorithm 34 implies $\exists j'$ $infoFromPredEvent_{j'}(p_2, p_1) \leq_{\mathcal{H}} getSucc_i(p_1, p_2) \wedge$ ($\forall o''' infoFromPredEvent_{j'}(p_2, p_1) \leq_{\mathcal{H}} o''' <_{\mathcal{H}} getSucc_i(p_1, p_2) \Rightarrow p_2 = succ_{\mathcal{H}_{o'''}}(p_1)$). Moreover, from API Ring History semantic requirement $n(ii)$, $infoFromPredEvent_{j'}(p_2, p_1) \in O_{\mathcal{H}} \Rightarrow infoForSuccEvent_{j'}(p_1) \wedge$ ($\forall o''' infoForSuccEvent_{j'}(p_1) \leq_{\mathcal{H}} o''' \leq_{\mathcal{H}} infoFromPredEvent_{j'}(p_2, p_1) \Rightarrow p_2 = succ_{\mathcal{H}_{o'''}}(p_1)$). Therefore, $\forall o''' infoForSuccEvent_{j'}(p_1) \leq_{\mathcal{H}} o''' \leq_{\mathcal{H}} getSucc_i(p_1, p_2) \Rightarrow p_2 = succ_{\mathcal{H}_{o'''}}(p_1)$.

  From API Ring History semantic requirement $n(iii)$, $\exists o''' infoForSuccEvent_{j'}(p_1) \leq_{\mathcal{H}} o''' \leq_{\mathcal{H}} o'' \wedge p_2 \neq succ_{\mathcal{H}_{o'''}}(p_1)$.

  Since $o'' <_{\mathcal{H}} getSucc_i(p_1, p_2)$, we have a contradiction.

  Hence any redistribute initiated by p_1 should have been initiated between o'' and o only after o'.

- Other two operations only change the low end of the range and hence are not relevant.

Let us now consider the operations that change $p_2.range$ between o' and o.

As shown above, o' is the only $infoFromPredEvent$ at p_2 between o'' and o. Therefore, the only operation which possibly changes $p_2.range.low$ is $recvRedistribute1_i(p_1)$ (because of a redistribute). We know that redistribute respects the invariant that $p_1.range.high = p_2.range.low$.

We therefore conclude that $p_1.range_{\mathcal{H}_o}.high = p_2.range_{\mathcal{H}_o}.low$.

We now argue that $p_1.range_{\mathcal{H}_{o_1}}.high = p_2.range_{\mathcal{H}_{o_2}}.low$

From Algorithm 30, there is a read lock on range at p_1. Therefore, $p_1.range_{\mathcal{H}_{o_1}} = p_1.range_{\mathcal{H}_o}$ and hence $p_1.range_{\mathcal{H}_{o_1}}.high = p_1.range_{\mathcal{H}_o}.high$.

Since there is a read lock on range at p_1, this rules out the possibility of a redistribute being initiated at p_1 which causes the range change at p_2 between o and o_2.

Since there is a read lock on range at operation $processScan1_i(p_2)$, it cannot happen that range changes because of a $infoFromPredEvent$ between $processScan1_i(p_2)$ and o_2.

Therefore, $p_2.range_{\mathcal{H}_{o_2}}.low = p_2.range_{\mathcal{H}_o}.low$.

Hence, we conclude that $p_1.range_{\mathcal{H}_{o_1}}.high = p_2.range_{\mathcal{H}_{o_2}}.low$

■

Claim 14 *Given a PEPPER Data Store History $\mathcal{H}$, $\forall p, p' \in \mathcal{P}$ ($\forall r'$ ($\forall i, k \in \mathcal{N}$ ($o = scanRange_{ik}(p, p', r') \in \mathcal{H} \Rightarrow range_{\mathcal{H}_o}(p) = p.range_{\mathcal{H}_o}$)))*

Proof: <Proof of this claim uses claim 15>

Let $o' = rangeChangeEvent(p, r, b)$ be the last range change event at p before o.

From claim 15, $range_{\mathcal{H}_{o'}}(p) = p.range_{\mathcal{H}_{o'}}$

From Algorithm 30, there is a read lock on $range$ at o.

From Algorithms 22, 23, 33, 34, 36, 37 and 41, every operation that modifies $p.range$ acquires a write lock on $p.range$ and throws up a RANGECHANGEEVENT event

before releasing the write lock. So, any modification of $p.range$ before $scanRange$ should be followed up by a RANGECHANGEEVENT event. Therefore, $p.range$ is not modified after o and so is $range(p)$.

Therefore, we conclude that $range_{\mathcal{H}_o}(p) = p.range_{\mathcal{H}_o}$

■

Claim 15 *Given a PEPPER Data Store History* $\mathcal{H}$, $\forall p \in \mathcal{P}$ ($\forall r$ ($\forall b$ ($\forall o \in O_{\mathcal{H}}$ ($o = rangeChangeEvent(p, r, b) \Rightarrow range_{\mathcal{H}_o}(p) = p.range_{\mathcal{H}_o}$)))).

Proof: The operations that modify $p.range$ are:

- $PinitFirstPeer1(p)$
- $initDS_1(p)$
- $insertHandler1_i(p)$
- $merge6_i(p)$
- $recvRedistribute1_i(p)$
- $recvMerge2(p)$
- $infoFromPred1_i(p)$

From Algorithms 22, 23, 33, 34, 36, 37 and 41, each of these operations is followed by a $rangeChangeEvent(p, p.range, b)$ operation. Moreover, in our implementation, every $rangeChangeEvent(p, r, b)$ operation in $\mathcal{H}$ must be preceded by one of the above 7 operations.

The result follows from the above observation.

■

12.2.5 $getLocalItems$ correctness

Theorem 3 *Any* PEPPER Data Store History $\mathcal{H}$ *satisfies getLocalItems correctness.*

Proof: <Proof of this theorem uses claim 16>

We need to show that: $\forall i \in \mathcal{N}$, $\forall p \in P$, $o = getLocalItems_i(p, j[]) \in O_{\mathcal{H}} \Rightarrow items_{\mathcal{H}_o}(p) = j[]$.

Note that $o = getLocalItems_i(p, j[]) \in O_{\mathcal{H}} \Rightarrow fail(p) \notin O_{\mathcal{H}_o}$.

Operations in $O_{\mathcal{H}}$ that change $p.items$ are: $O_{I\mathcal{H}p} = \{PinitFirstPeer1(p), initDS1(p), insertItem4(p, i), deleteItem4(p, i), insertHandler1(p, p'), merge6_i(p), recvRedistribute1_i(p), recvMerge2(p), initInsertLocalItem1(p, j), initDeleteLocalItem1(p, j)\}$

From the locks held on $range$ and $items$ in the implementation, $o_1 \in O_{I\mathcal{H}p}$ iff there is $o' \in O_{I\mathcal{H}}(p)$ such that $o_1 \leq_{\mathcal{H}} o'$ and no other $o_2 \in O_{I\mathcal{H}p}$ occurs between o_1 and o'. Moreover, since $o = getLocalItems$ acquires a read lock on $items$, o cannot occur between o_1 and o'.

From claim 16, $o' \in O_{I\mathcal{H}} \Rightarrow p.items_{\mathcal{H}_{o'}} = items_{\mathcal{H}_{o'}}(p)$.

Therefore, correctness of getLocalItems follows.

■

Claim 16 *Given a PEPPER Data Store History* $\mathcal{H}$, $\forall p \in P_{\mathcal{H}}$ ($o \in O_{I\mathcal{H}} \Rightarrow p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$).

Proof: We prove this by induction on the operations in $\mathcal{H}$.

- *Base case*: Smallest valid PEPPER Data Store History $\mathcal{H}$ contains all operations from $initFirstPeer(p)$ to $firstPeer(p)$ in Algorithm 22. $o = rangeChangeEvent(p, r, b)$ is the only operation in $O_{\mathcal{H}}$ that belongs to $O_{I\mathcal{H}}$. Also, $\forall p_1 \in \mathcal{P}$ ($items_{\mathcal{H}_o}(p_1) = \{\} = p_1.items_{\mathcal{H}_o}$) and $items_{\mathcal{H}}(\phi) = \{\}$.

- *Induction Hypothesis*: We assume that the induction hypothesis holds for all $\mathcal{H}'$ such that $|O_{\mathcal{H}'}| = k$.

- *Induction Step*: Now consider $\mathcal{H}$ such that $|O_{\mathcal{H}}| = k + 1$. Consider an op $o \in O_{\mathcal{H}}$ such that $\nexists o' \in \mathcal{H}\ o <_{\mathcal{H}} o'$ (Note that there should exist one such operation *o*). $\mathcal{H}' = (O_{\mathcal{H}} - \{o\}, \leq_{\mathcal{H}'})$, where $o_1 \leq_{\mathcal{H}'} o_2$ iff $o_1, o_2 \in O_{\mathcal{H}} - \{o\} \wedge o_1 \leq_{\mathcal{H}} o_2$, is a PEPPER Data Store History. Since $|\mathcal{H}'| = k$, by induction hypothesis, $items_{\mathcal{H}'}$ is defined.

  Operations that affect $items_{\mathcal{H}}(p)$ are in $O_{I\mathcal{H}}(p) = \{insertedItem(p, p', j), deletedItem(p, p', j), insertLocalItem(p, j), deleteLocalItem(p, j), rangeChangeEvent(p, r, b), fail(p)\}$.

  If $o \notin O_{I\mathcal{H}}(p)$, the claim holds by induction hypothesis.

  Let us now consider different possibilities for *o*.

  – $o = fail(p)$. $\forall p_1 \neq p$ ($p_1.items_{\mathcal{H}} = p_1.items_{\mathcal{H}'}$). From induction hypothesis, $p_1.items_{\mathcal{H}'} = items_{\mathcal{H}'}(p_1)$, so $p_1.items_{\mathcal{H}} = items_{\mathcal{H}'}(p_1)$. From the definition of $items()$, $items_{\mathcal{H}}(p_1) = items_{\mathcal{H}}(p_1)$. So, $\forall p_1 \neq p$ ($p_1.items_{\mathcal{H}} = items_{\mathcal{H}}(p_1)$).

  For failed peer *p*, we define $p.items_{\mathcal{H}} = \phi$ (we do not consider the items in failed peers). Therefore, $items_{\mathcal{H}}(p) = p.items_{\mathcal{H}}$.

  Also, $items_{\mathcal{H}}(\phi) = items_{\mathcal{H}'}(\phi) - \{i | i \in items_{\mathcal{H}'}(\phi) \wedge i.skv \in range_{\mathcal{H}_o}(p)\}$.

  – $o = rangeChangeEvent(p, r, b)$: In our implementation, operation $o = rangeChangeEvent(p, r, b)$ occurs in the following algorithms – 22, 33, 23, 34, 36, 36 and 41.

  Consider o in Algorithm 22. $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o} = \{\}$.

  Consider o in Algorithm 33. Since $p.range$ is split and so is $p.items$, $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$. Also, note that split items that will be sent to the successor are now in $items_{\mathcal{H}_o}(\phi)$.

  Consider o in Algorithm 23. From API Ring history restrictions, $inserted(p) \in O_{\mathcal{H}} \Rightarrow \exists p' \in \mathcal{P}\ insert(p', p) \leq_{\mathcal{H}} inserted(p)$. Therefore, from Algorithm 33, $o' = rangeChangeEvent(p', r', b') \in O_{\mathcal{H}}$.

  In Algorithm 33, range of p' is split into two ranges r' and r. Items put in $items_{\mathcal{H}_{o'}}(\phi)$ are the ones used to set $p.items$ in Algorithm 23. Therefore, $p.items_{\mathcal{H}_{o'}} = items_{\mathcal{H}_{o'}}(p)$ and $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$

  Consider o in Algorithm 36. Since $p.range$ is split and so is $p.items$, $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$. Also, note that split items that will be sent to the predecessor are now in $items_{\mathcal{H}_o}(\phi)$.

  Consider o in Algorithm 23. From Algorithm 22, $\neg\ initfirstpeer(p) \Rightarrow \neg\ initRing(p)$. Therefore, from API Ring History semantic requirement f, $inserted(p) \in O_{\mathcal{H}} \Rightarrow \exists p' \in \mathcal{P}\ insert(p', p) \leq_{\mathcal{H}} inserted(p)$. Therefore, from Algorithm 33, $o' = rangeChangeEvent(p', r', b') \in O_{\mathcal{H}}$.

  Here, range of p' is split into two ranges r' and r. At o', items put in $items_{\mathcal{H}_{o'}}(\phi)$ are the ones used to set $p.items$ in Algorithm 23. Therefore, $p.items_{\mathcal{H}_{o'}} = items_{\mathcal{H}_{o'}}(p)$ and $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$

  Consider o in Algorithm 36. Since $p.range$ is split and so is $p.items$, $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$. Also, note that split items that will be sent to the predecessor are now in $items_{\mathcal{H}_o}(\phi)$.

  Consider o in Algorithm 37 Since $p.range$ is set to ϕ and $p.items$ is set to $\{\}$, $p.items_{\mathcal{H}_o} =$

$items_{\mathcal{H}_o}(p)$. Also, note that items that will be sent to the predecessor are now in $items_{\mathcal{H}_o}(\phi)$.

Consider o in Algorithm 34. $p.range$ is extended and $p.items$ is set to include items from successor in the extended range. From implementation of redistribute and merge, $o \in \mathcal{H} \Rightarrow \exists p' \in \mathcal{P}$ ($o' = rangeChangeEvent(p', r', b') \leq_{\mathcal{H}} o \wedge r'.low = r.high$). As we have seen in the above two possible cases (redistribute and merge), items from p' are in $items_{\mathcal{H}'_o}(\phi)$. Therefore, $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$.

Consider o in Algorithm 41. This operation occurs because the predecessor of p (say p') failed and hence $p.range$ is extended to cover what was p''s range. In this case, the items in p' are lost. Note that $o' = fail(p')$ operation sets $items_{\mathcal{H}_{o'}}(p')$ to $\{\}$. Therefore, no new items are inserted into $p.items$.

Using induction hypothesis, we conclude the result in this case ($o = rangeChangeEvent(p, r, b)$).

– $o = insertedItem(p, p', j)$

In this case, $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) \cup \{j\}$. Let o' be the last opertion in $O_{I\mathcal{H}}(p)$ before o. From induction hypothesis, $items_{\mathcal{H}_{o'}}(p) = p.items_{\mathcal{H}_{o'}}$.

From the locks held on $range$ and $items$ in the implementation, $o_1 \in O_{I\mathcal{H}p}$ iff there is $o'' \in O_{I\mathcal{H}}(p)$ such that $o_1 \leq_{\mathcal{H}} o''$ and no other $o_2 \in O_{I\mathcal{H}p}$ occurs between o_1 and o''.

Therefore, from Algorithm 27, $insertItem4(p, j)$ is the only operation after o' and before o that modified $p.items$. Hence, $p.items_{\mathcal{H}} = p.items_{\mathcal{H}_o} = p.items_{\mathcal{H}_{o'}} \cup \{j\}$

Therefore, $items_{\mathcal{H}_o}(p) = p.items_{\mathcal{H}_o}$.

– $o = insertLocalItem(p, j)$

In this case, $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) \cup \{j\}$. Let o' be the last opertion in $O_{I\mathcal{H}}(p)$ before o. From induction hypothesis, $items_{\mathcal{H}_{o'}}(p) = p.items_{\mathcal{H}_{o'}}$.

From the locks held on $range$ and $items$ in the implementation, $o_1 \in O_{I\mathcal{H}p}$ iff there is $o'' \in O_{I\mathcal{H}}(p)$ such that $o_1 \leq_{\mathcal{H}} o''$ and no other $o_2 \in O_{I\mathcal{H}p}$ occurs between o_1 and o''.

Therefore, from Algorithm 43, $initInsertLocalItem1(p, j)$ is the only operation after o' and before o that modified $p.items$. Hence, $p.items_{\mathcal{H}} = p.items_{\mathcal{H}_o} = p.items_{\mathcal{H}_{o'}} \cup \{j\}$

Therefore, $items_{\mathcal{H}_o}(p) = p.items_{\mathcal{H}_o}$.

– $o = deletedItem(p, p', j)$

In this case, $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) - \{j\}$. Let o' be the last opertion in $O_{I\mathcal{H}}(p)$ before o. From induction hypothesis, $items_{\mathcal{H}_{o'}}(p) = p.items_{\mathcal{H}_{o'}}$.

From the locks held on $range$ and $items$ in the implementation, $o_1 \in O_{I\mathcal{H}p}$ iff there is $o'' \in O_{I\mathcal{H}}(p)$ such that $o_1 \leq_{\mathcal{H}} o''$ and no other $o_2 \in O_{I\mathcal{H}p}$ occurs between o_1 and o''.

Therefore, from Algorithm 28, $deleteItem4(p, j)$ is the only operation after o' and before o that modified $p.items$. Hence, $p.items_{\mathcal{H}} = p.items_{\mathcal{H}_o} = p.items_{\mathcal{H}_{o'}} \cup \{j\}$

Therefore, $items_{\mathcal{H}_o}(p) = p.items_{\mathcal{H}_o}$.

– $o = deleteLocalItem(p, j)$

In this case, $items_{\mathcal{H}}(p) = items_{\mathcal{H}'}(p) - \{j\}$. Let o' be the last operation in $O_{I\mathcal{H}}(p)$ before o. From induction hypothesis, $items_{\mathcal{H}_{o'}}(p) = p.items_{\mathcal{H}_{o'}}$.

From the locks held on $range$ and $items$ in the implementation, $o_1 \in O_{I\mathcal{H}p}$ iff there is $o'' \in O_{I\mathcal{H}}(p)$ such that $o_1 \leq_{\mathcal{H}} o''$ and no other $o_2 \in O_{I\mathcal{H}p}$ occurs between o_1 and o''.

Therefore, from Algorithm 44, $initDeleteLocalItem1(p, j)$ is the only operation after o' and before o that modified $p.items$. Hence, $p.items_{\mathcal{H}} = p.items_{\mathcal{H}_o} = p.items_{\mathcal{H}_{o'}} - \{j\}$

Therefore, $items_{\mathcal{H}_o}(p) = p.items_{\mathcal{H}_o}$.

■

Corollary: Given a PEPPER Data Store History $\mathcal{H}$, $items_{\mathcal{H}}$ is well-defined.

Algorithm 45 : $p.initRangeQuery_i(lb, ub)$

```
⌈ 0: // initiate a scanRange
initRangeQuery_i(p)
⌊ 1: p.initScanRange_i(lb, ub, rangeQueryHandlerId, <
     p, φ >)
⌈ 2: on receiving all results
rangeQuery_i(p)
⌊ 3: raiseEvent(RANGEQUERY, results)
```

Algorithm 46 : $p.rangeQueryHandler_{ik}(r_k, < pid, resultSoFar >)$

```
⌈ 0: // Get items from p's Data Store
| 1: readLock items
rangeQueryHandler1_i(p)
| 2: find qitems from p.items in range r_k
⌊ 3: releaseLock items
⌈ 4: send < qitems, r_k > to peer pid
sendResults_i(p, pid)
⌊ 5:
```

12.2.6 Query correctness In this section, we prove that the PEPPER Data Store implementation returns correct range query results. Before we state and prove the theorem, we first introduce the notion of a *live* item.

Definition 16 (Live Item) *An item i is* live *in API Data Store History $\mathcal{H}$, denoted by $live_{\mathcal{H}}(i)$, iff $\exists p \in P\ i \in items_{\mathcal{H}}(p)$.*

Theorem 4 *Given a PEPPER Data Store History $\mathcal{PH}$, all query results produced in $\mathcal{PH}$ are* correct *i.e. a set R of items is a* correct query result *for a query Q initiated at some peer p with operation $o_s = initRangeQuery_i(p)$ and successfully completed with operation $o_e = rangeQuery_i(p)$ iff the following two conditions hold:*

1. $\forall i \in R$ ($satisfies_Q(i) \wedge \exists o$ ($o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i)$))
2. $\forall i$ ($\forall o$ ($satisfies_Q(i) \wedge o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i) \Rightarrow i \in R$)).

Proof: <Proof of this theorem uses theorem 2>

Consider $i \in R$. Suppose item i was included in the result because it was sent as part of the operation $sendResults_i(p, pid)$ at some peer p. From Algorithm 46, $sendResults_i(p, pid) \in \mathcal{H} \Rightarrow o = rangeQueryHandler1_i(p) \leq_{\mathcal{H}} sendResults_i(p, pid)$. Note that $o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e$.

We argue that $live_{\mathcal{H}_o}(i)$. From Algorithm 46, $i \in p.items_{\mathcal{H}_o}$. Using the argument for getLocalItems correctness, $p.items_{\mathcal{H}_o} = items_{\mathcal{H}_o}(p)$. Therefore, $i \in items_{\mathcal{H}_o}(p)$ and hence $live_{\mathcal{H}_o}(i)$. This proves condition 1.

Consider i such that $satisfies_Q(i) \Rightarrow i.skv \in [lb, ub]$ and $\forall o\ o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i)$. Let j be the index of the $initScanRange$ invocation in o_s. Let $i.skv \in r$, for some $r \in rangeSet(j)$ (from scanRange correctness there exists exactly one such r). Let p be the peer corresponding to r. We also know that $o' = rangeQueryHandler1_i(p) \Rightarrow o_s \leq_{\mathcal{H}} o' \leq_{\mathcal{H}} o_e$. Therefore, $live_{\mathcal{H}_{o'}}(i)$. Hence, $i \in items_{\mathcal{H}_{o'}}(p)$. Using the argument for getLocalItems correctness, $items_{\mathcal{H}_{o'}}(p) = p.items_{\mathcal{H}_{o'}}$. So, $i \in p.items_{\mathcal{H}_{o'}}$. Therefore, from Algorithm 46, $i \in R$. This proves condition 2.

■

12.2.7 Main Result

Theorem 5 *Given a PEPPER Data Store History $\mathcal{PH}$, $\Pi_{O_d(\mathcal{P})}(\mathcal{PH})$ is an API Data Store History.*

Proof: We show that $\Pi_{O_d(\mathcal{P})}(\mathcal{PH}) = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$ satisfies all the sematic requirements.

- (Any operation other than fail involving p *happened before* $fail(p)$.)

  $\forall p \in \mathcal{P}$ ($fail(p) \in O_{\mathcal{H}} \Rightarrow$ ($\forall op(p) \in \mathcal{O}_d(p)$ ($op(p) \in O_{\mathcal{H}} \wedge op(p) \neq fail(p) \Rightarrow op(p) \leq_{\mathcal{H}} fail(p)$)) $\wedge$ ($\forall p' \in \mathcal{P}, \forall op(p,p') \in \mathcal{O}_d(p,p')$ ($op(p,p') \in O_{\mathcal{H}} \Rightarrow op(p,p') \leq_{\mathcal{H}} fail(p)$))).

  Follows from happened before constraints.

- (A scanRange operation should be initiated before it is completed.)

  $\forall p,p' \in \mathcal{P}$ ($\forall i,k \in \mathcal{N}$ ($\forall r' \in \mathcal{N} \times \mathcal{N}$ ($scanRange_{ik}(p',p,r') \in O_{\mathcal{H}} \Rightarrow \exists lb,ub \in \mathcal{N}$ ($initScanRange_i(p,lb,ub) \in O_{\mathcal{H}} \wedge initScanRange_i(p,lb,ub) \leq_{\mathcal{H}} scanRange_{ik}(p',r')$)))).

  $\forall p \in \mathcal{P}$ ($\forall i,lb,ub \in \mathcal{N}$ ($doneScanRange_i(p,lb,ub) \in O_{\mathcal{H}} \Rightarrow initScanRange(p,lb,ub) \leq_{\mathcal{H}} doneScanRange_i(p,lb,ub)$)).

  Follows from theorem 2.

- (An insertItem operation should be initiated before it is completed.)

  $\forall p \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($insertItem(p,j) \in O_{\mathcal{H}} \Rightarrow initInsertItem(p,j) \leq_{\mathcal{H}} insertItem(p,j)$)).

  Follows from Algorithm 24.

- (An insertedItem operation should be initiated before it is completed.)

  $\forall p,p' \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($insertedItem(p',p,j) \in O_{\mathcal{H}} \Rightarrow initInsertItem(p,j) \leq_{\mathcal{H}} insertedItem(p',p,j)$)).

  Follows from Algorithms 24 and 27.

- (An deleteItem operation should be initiated before it is completed.)

  $\forall p \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($deleteItem(p,j) \in O_{\mathcal{H}} \Rightarrow initDeleteItem(p,j) \leq_{\mathcal{H}} deleteItem(p,j)$)).

  Follows from Algorithm 25.

- (An deletedItem operation should be initiated before it is completed.)

  $\forall p,p' \in \mathcal{P}$ ($\forall j \in \mathcal{T}$ ($deletedItem(p',p,j) \in O_{\mathcal{H}} \Rightarrow initDeleteItem(p,j) \leq_{\mathcal{H}} deletedItem(p',p,j)$)).

  Follows from Algorithms 25 and 28.

- (A getLocalItems operation should be initiated before it is completed.)

  $\forall p \in \mathcal{P}$ ($\forall i \in \mathcal{N}$ ($\forall j \in \mathcal{T}$ ($getLocalItems_i(p,j[]) \in O_{\mathcal{H}} \Rightarrow initGetLocalItems_i(p) \leq_{\mathcal{H}} getLocalItems_i(p,j[])$)))

  Follows from Algorithm 42.

- (`DSINFOFROMPREDEVENT` at peer p implies `DSINFOFORSUCCEVENT` occurred at the some peer p')

  $\forall p \in \mathcal{P}$ ($dsInfoFromPredEvent(p) \in O_{\mathcal{H}} \Rightarrow \exists p' \in \mathcal{P}$ ($dsInfoForSuccEvent(p',p) \leq_{\mathcal{H}} dsInfoFromPredEvent(p)$))

  From Algorithm 23, $\forall p \in \mathcal{P}$ ($dsInfoFromPredEvent(p) \in O_{\mathcal{H}} \Rightarrow inserted(p) \leq_{\mathcal{H}} dsInfoFromPredEvent(p) \wedge \neg(initFirstPeer(p))$)

  From Algorithm 22, $\neg\, initFirstPeer(p) \in O_{\mathcal{H}} \Rightarrow \neg\, initRing(p) \in O_{\mathcal{H}}$

  Hence, from API Ring History semantic requirement (f), $inserted(p) \in O_{\mathcal{H}} \Rightarrow \exists p' \in \mathcal{P}$ ($insert(p',p) \leq_{\mathcal{H}} inserted(p)$).

  From Algorithm 33 $\forall p,p' \in \mathcal{P}$ ($insert(p',p) \in O_{\mathcal{H}} \Rightarrow dsInfoForSuccEvent(p',p) \leq_{\mathcal{H}} insert(p',p)$).

  Hence, $\forall p \in \mathcal{P}$ ($dsInfoFromPredEvent(p) \in O_{\mathcal{H}} \Rightarrow \exists p' \in \mathcal{P}$ ($dsInfoForSuccEvent(p',p) \leq_{\mathcal{H}} dsInfoFromPredEvent(p)$))

- (Successive range changes are such that one end of the range does not change.) $\forall p \in \mathcal{P}$ ($o = rangeChangeEvent(p,r,b) \in O_{\mathcal{H}} \wedge o' = rangeChangeEvent(p,r',b') \in O_{\mathcal{H}} \wedge r' \neq \phi \wedge \nexists r''$ ($o'' = rangeChangeEvent(p,r'',b'') \in O_{\mathcal{H}} \wedge o <_{\mathcal{H}} o'' <_{\mathcal{H}} o'$) $\Rightarrow r.low = r'.low \vee r'.low = r'.high$)

  After initialization, the only operations which change $p.range$ are:

  1. $insertHandler1_i(p_1)$ (on a split, at the splitting peer): This operation only changes the high end of the range.
  2. $merge6_i(p_1)$ (on the peer which has initiated a merge/redistribute): This operation only changes the high end of the range.
  3. $recvRedistribute1_i(p_1)$ (on the peer which is redistributing based on the predecessor's request): This operation only changes the low end of the range.
  4. $recvMerge2(p_1)$: Sets range to ϕ.
  5. $infoFromPred1_i(p_1)$ (on receiving the infoFromPredEvent): This operation only changes the low end of the range.

Note that there cannot be a range change event at p_1 after the one corresponding to $recvMerge2(p_1)$ which sets range to ϕ. Other than $recvMerge2(p_1)$, note that each of the above operations modifies only one end of the range. The result therefore follows.

- $items_{\mathcal{H}}$ is well-defined

  Follows from claim 16.

- Semantics of scanRange

  $\forall i \in \mathcal{N}\ \forall lb, ub\ \forall p_1 \in \mathcal{P}\ o_e = doneScanRange_i(p_1, lb, ub) \in O_{\mathcal{H}} \Rightarrow$

  1. $o_s = initScanRange_i(p_1, lb, ub) \leq_{\mathcal{H}} o_e$
  2. $\forall o \in scanOps(i)\ \forall p\ \forall r\ o = scanRange_i(p, p_1, r) \Rightarrow o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge r \subseteq range_{\mathcal{H}_o}(p)$
  3. $\forall o_l, o_m \in scanOps(i)\ o_l \neq o_m \wedge \forall p_l, p_m\ \forall r_l, r_m\ o_l = scanRange_i(p_l, p_1, r_l) \wedge o_m = scanRange_i(p_m, p_1, r_m) \Rightarrow \neg(o_l \bowtie o_m)$
  4. $[lb, ub] = \cup_{r \in rangeSet(i)}(r)$

  scanRange correctness follows from theorem 2.

- Semantics of getLocalItems

  $\forall i \in \mathcal{N}\ (\ \forall j \in \mathcal{T}\ (\ \forall p \in P\ (\ o = getLocalItems_i(p, j[]) \in O_{\mathcal{H}} \Rightarrow items_{\mathcal{H}_o}(p) = j.$

  Follows from theorem 3.

■