Synoptic : Unraveling Distributed Systems with Message-level Summarization and Inference Techniques

Slava Chernyak Josh Goodwin Sigurd Schneider Ivan Beschastnikh

University of Washington
Computer Science and Engineering
{*chernyak, dravir, sigurd, ivan*}*@cs.washington.edu*

Abstract

Distributed system implementations are difficult to debug and understand. Often the only way to gain insight into a system is to track messages sent between nodes, which may be implementing multiple concurrently executing distributed protocols. There are few tools to help with this task – a common approach is to log all node events and then manually inspect the logs for clues that might reveal why the system behaves the way that it does.

To help with this task we developed *Synoptic* – a summarization tool that takes the system's logged messages as input and outputs a finite state automata representation. Compared to the raw message traces, the resulting representation is visually concise, and is also rich with inferred temporal relationships mined from the messages.

We describe how Synoptic works, and present its core algorithms and the intuition that led to their design. We evaluate Synoptic by applying it to a variety of real and synthetic distributed systems' traces. We evaluate its performance with benchmarks, and we carry out a user study with a distributed systems developer to evaluate its utility. Our results suggest that Synoptic has reasonable overhead for an offline analysis tool, and that it helps to augment distributed system designers understanding of system behavior.

1 Introduction

Distributed systems are often difficult to debug and to understand. This difficulty can be usually ascribed to the complex behavior of the nodes comprising the system. For instance, nodes may execute intertwined and difficult to follow distributed protocols. Another difficulty is the distributed nature of state. No node can observe all the system state, and due to asynchrony in the networking environment, observations of remote state are stale unless protocols with significant overhead are used.

Developers who design and implement distributed systems often analyze system behavior by collecting execution traces. However, even a small system may generate thousands of messages – just a few nodes running a standard distributed protocol such as two-phase commit [17], or Paxos [20] can generate this many messages over just a few minutes of execution time. Manual inspection of such traces scales poorly and by skimming over the trace, a developer may easily miss an important system behavior. It is therefore challenging to understand the behavior of a system from collected traces without tools to support this process. In this paper we describe the design and implementation of *Synoptic* – a tool to generate a Finite State Machine (FSM) representation of distributed system executions. Synoptic analyzes the network-level traces generated by nodes in a distributed system and can help the developer to understand actual system behavior and help to uncover potential system correctness and performance issues.

We approach the problem of understanding the system as a summarization and a data reduction challenge. A Synoptic-derived representation is not a specification for a system. But the representation is a compact, and most importantly useful, summary of the observed system behaviors. Synoptic describes the behavior of a distributed system while preserving certain temporal invariants mined from the the ordering of messages in the trace.

As part of building Synoptic we developed a framework for searching through the space of potential FSM representations of the initial system trace. We develop two approaches to explore this space – coarsening and refinement. Coarsening starts out with the concrete trace and attempts to compact the representation at each step without admitting a representation that may represent illegal traces. Refinement is the reverse process, in which a representation is expanded starting from a single state until it admits all the observed system traces. To guide coarsening and refinement we develop a decision engine

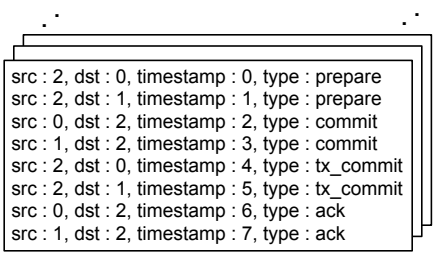


Figure 1: A partial example trace of a three node run of the two-phase commit protocol.

and a set of policies relevant to our domain. Within this framework we implement two concrete algorithms that repeatedly apply one of the exploration strategies. Our coarsening algorithm, *GK-Tail*, is based on the algorithm developed by Lorenzoli, et. al. [22] relying on the intuition that messages between nodes may be considered as remote method invocations. Our refinement algorithm, *Bikon*, is a bisimulation-inspired algorithm.

We applied Synoptic to real and synthetic traces of distributed systems. We evaluated the derived representations in a user study with a distributed systems developer. In this study, the developer was asked to talk through his understanding of the represented system behavior. From this study we found that Synoptic's output matched the mental model of the developer and that the developer was able to remember specific corner-case design choices by studying the representation.

In summary, the contributions of this paper are the following. We develop a framework for searching through the space of potential FSM representation of a distributed system trace. Within this framework we implement two algorithms, and evaluate the resulting Synoptic tool on a variety of distributed system traces and in the context of a user study. This paper includes the representations found by both algorithms for a number of popular distributed protocols and systems, the results of our user study, and performance results for our coarsening and refinement algorithms.

In the next section we motivate our work with an example that illustrates how Synoptic may be used to understand a complex distributed protocol. We then overview our assumptions and the design of our framework in sections 3 and 4. We then describe the interface to our decision engine and some of the policies it implements in section 5. We follow with section 6 and 7 in which we describe our two algorithms – the adapted GK-Tail and Bikon. Then, we present an evaluation of Synoptic in section 8. We finish with a discussion (section 9), an overview of the related work (section 10), and conclude with section 11.

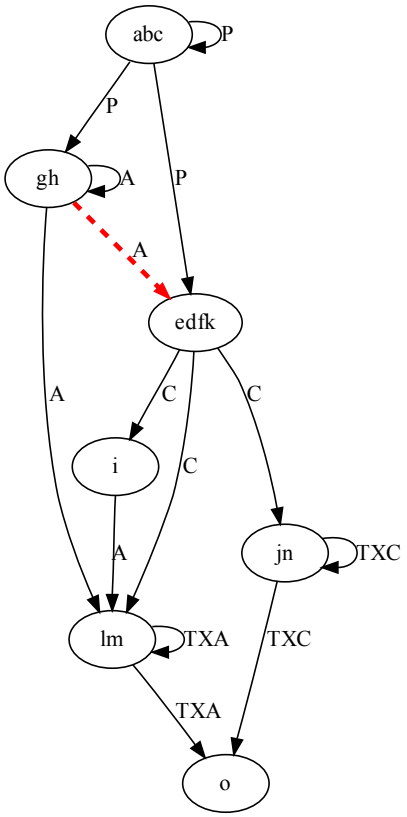


Figure 2: A Synoptic-generated representation of a buggy two-phase commit protocol trace for two nodes, with one transaction manager. Transitions represent messages sent/received by a replica; states represent abstract system states. States are labeled with letters whose combination indicates a particular merge of single letter states of the FSM for the non-buggy version of the system in Figure 11. The red dotted edge indicates the invalid transition.

2 Motivation

To motivate the problem more concretely we present an example scenario in which the Synoptic representation of a system trace might help a developer understand system behavior. We study an atomic commitment protocol called two-phase commit [17], which is widely used to implement distributed atomic transactions. This protocol is non-trivial to debug in large systems because it is inherently concurrent and composed of multiple stages.

2.1 A Two-Phase Commit Example

The two-phase commit protocol must adhere to a number of invariants, which range over protocol messages – (P)repare, (C)ommit, (A)bort, (TXA)bort, (TXC)ommit. For a system of three nodes (with one node acting as the transaction manager) Table 1 lists some of the basic in-

Invariants	Explanation
$\Box(A \rightarrow \Diamond TXA)$	An abort by even one replica must prompt an eventual transaction abort.
$\Box(A \rightarrow \Box\neg TXC)$	A transaction cannot be committed if any replica aborts.
$\Box(TXC \rightarrow \Box\neg TXA)$ $\Box(TXA \rightarrow \Box\neg TXC)$	A transaction commit and transaction abort are mutually exclusive – only one can ever be observed.
$\|C\| + \|A\| = 2$	With two-replicas, the sum of commit and abort message counts must be two.

Table 1: Some two-phase commit invariants in LTL notation. In LTL, $\Box$ and $\Diamond$ mean *for all time*, and *eventually* respectively.

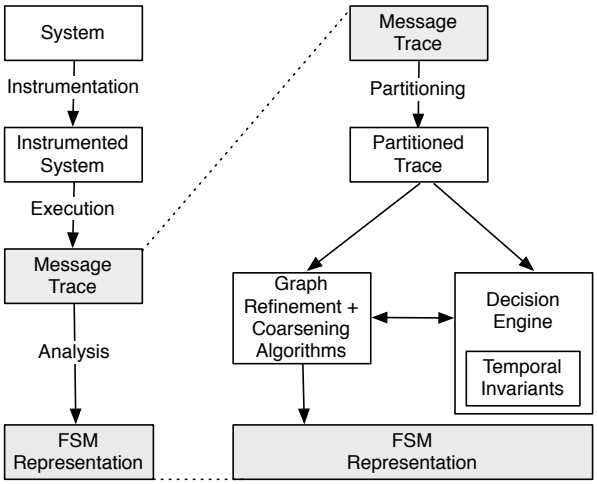


Figure 3: An overview of the system pipeline (left) and the details of the analysis pipelines implementing (right).

variants.

Given a trace that looks like Figure 1, how can a developer easily verify whether the invariants in Table 1 hold? Figure 2 shows the representation generated by Synoptic for a buggy version of the system, while Figure 11 (in section 8) shows the representation for a correct system. It is easy to verify by inspection that all of the above invariants are maintained in the correct representation. Moreover, the representation of the buggy system hints at where a problem might lie – the second invariant in the Table is violated since in Figure 2 an A message precedes a TXC message (the red, dotted edge in the Figure).

Note that the two-phase commit protocol also has non-temporal, or structural invariants (e.g. the last invariant in Table 1). Currently Synoptic does not support such invariants.

3 Assumptions

We make a few assumptions concerning the environment or the distributed system whose traces we analyze.

- We assume a non-broadcast networking medium at the message capture layer. For example, we do not handle wireless environments in which the nodes in the system sniff and thereby receive all messages. However, we can operate in a wireless environment with all nodes restricted to using TCP for communication, or in which all sniffed traffic is also logged in the trace.

- We assume that the nodes are using a reliable communication protocol so that messages are never lost. This assumption may be dropped if in addition to message transmission the trace also includes message reception. In the absence of observable reception, Synoptic cannot distinguish between messaging behavior at the receiver node caused by a lost message and receiver behavior caused by message reception.

- We assume that messages reordering does not occur. This is necessary to prevent re-orderings from adversely impacting our temporal invariant inference algorithm. A more robust algorithm can relax this assumption to a certain degree.

- If the system is composed of just one physical host the local clock can be used for timestamping all messages. To analyze the behavior of multiple network nodes, a consistent notion of time is necessary. As a simplification, we assume correctly synchronized clocks (e.g. nodes use NTP). This assumption can be relaxed by establishing a global partial ordering on all the messages in the system with a vector clock algorithm [19].

4 System Design

Synoptic is comprised of several components. Figure 3 illustrates these components, and in the rest of this section we briefly overview each of them.

Synoptic starts by inferring a set of temporal properties between messages in the trace. Synoptic then runs either the coarsening or the refinement algorithm to derive a FSM representation of the trace, which is presented to the user.

4.1 Message Capture

To be used with Synoptic, the input system must be manually modified to capture all network-level messages. We use the term *message* in a general sense to refer to a formatted message sent between two nodes in a distributed system.

There are several ways to capture messages (section 9 discusses some alternatives). The core of Synoptic is independent of how the messages are captured, and in our evaluation (section 8) we study systems whose messages are captured in a variety of ways. We assume that the the end result of message capture process is a trace that has the following structure:

$$\langle \text{src}, \text{dst}, \text{timestamp}, [datafields] \rangle$$

For example a three way handshake trace could look like the following sequence of six messages:

$$\begin{aligned} \langle 1, 2, 0, [REQ] \rangle \\ \langle 2, 1, 1, [RESP] \rangle \\ \langle 1, 2, 2, [ACK] \rangle \\ \langle 3, 2, 3, [REQ] \rangle \\ \langle 2, 3, 4, [RESP] \rangle \\ \langle 3, 2, 5, [ACK] \rangle \end{aligned}$$

Each message must include extra meta information such as the source and destination node addresses above. Because we assume that messages are never lost (see section 3), we only capture outgoing messages. The node traces are then manually collected and manually merged into a single trace. To make this feasible, we assume that the nodes use NTP or a similar protocol to synchronize their local time.

4.2 Analysis

The analysis component is the core of Synoptic. To perform analysis, the user must first manually create semantically meaningful partitions of the message trace. Second, the user must specify one of two algorithms Synoptic should use – GK-Tail, or Bikon. Depending on the algorithm, Synoptic then converts the input message trace into one of two FSM representations. The first representations is more direct as it makes no assumption about the structure of the underlying process. The second representation is more intuitive to systems designers. Either of the representations can be converted to the other.

Next, Synoptic explores the set of possible FSM representations until it finds one that satisfies certain constraints. Depending on the algorithm, this exploration takes the form of either refinement or coarsening of the FSM. Coarsening starts with a concrete representation of the partitioned trace and merges states to derive smaller representations. Refinement starts with a single state representing all messages of the same type in a partition trace, and enlarges the representations by splitting these states. Finally, the end result of this exploration is displayed to the user as an FSM image.

4.2.1 Trace Partitioning

The final manual step performed by the user is trace partitioning. This groups messages into instances of the process that generated the messages. Partitioning is necessary to limit the scope relations between messages in the trace. Its also necessary to tease apart concurrent protocols. As a general rule, messages that are generated independently should belong to different partitions.

The choice of partition function is crucial to derive a proper representation for the traces. For example, the two-phase commit protocol traces used in the evaluation are partitioned by run of the system – each complete execution of the protocol is a unique trace. Another partitioning scheme might partition the global trace by node, so that the resulting representation captures a single node's perspective of the system.

For this step we use a set of partitioning predicates $p_1, ..., p_n$, such that for every message exactly one predicate is true. If predicate p_i is true for a message m, we say message m has type i. The partitioning predicates are assumed to be supplied by the user.

4.2.2 Interpreting Traces as Graphs

Synoptic employs two different models to interpret the partitioned messages as directed graphs. The first assumes as little as possible about the process that generated the messages, and relates messages according to their properties. The second model considers the messages as being generated by a system that may or may not transition between different states each time a message is sent or received. As we will show, the representations of these models are equivalent. Next we formally define these representations, and show them to be equivalent.

Messages are temporally related. The first model captures how message relate according to their properties. The relation we use is time (i.e. message X follows message Y). However, there are other relations such as is-response (i.e. message X is a response to message Y). Given a trace of messages M, Synoptic constructs a graph $G = (V, E)$, in which vertices represent sets of messages:

$$V = \mathcal{P}(M)$$

We choose sets of messages to allow flexibility that we will need later on. For now, we only relate singleton sets of vertices according to the the time relation t:

$$E = \{(\{m_1\}, \{m_2\}, t) \mid m_1, m_2 \in M \wedge m_1.tstamp < m_2.tstamp\}$$

Messages transform System State The second model associates each message with a system state that produced the message. Given a message trace M, Synoptic constructs a graph $G = (V, E)$ in which every vertex represents some notion of *system state*. Since we assume the input trace to be linear, we need $|M| + 1$ states. We can use the natural numbers $[1, |M|+1]$ as states, but for flexibility we use sets of natural numbers:

$$V = \{\{i\} \mid i \in [1, |M| + 1]\}$$

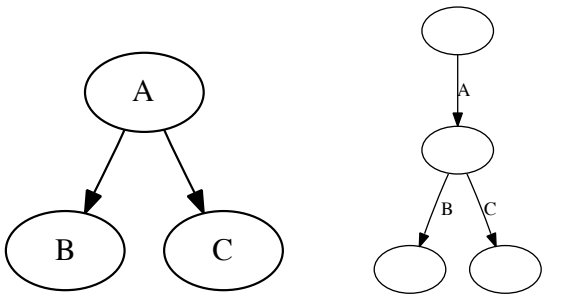
We now want to relate states according to the messages that cause the transition. The fundamental assumption we make here is that if message m_1 occurred immediately before message m_2 in the trace, then m_1 caused the system to enter the state associated with m_2. We capture this by interpreting m_1 as a transition from the system state associated with m_1 to the system state associated with m_2. Since we required the `tstamp` fields to be distinct in every trace, ordering the set $\{m.tstamp \mid m \in M\}$ yields a sequence $s_1, \ldots, s_{|M|}$ of length $|M|$. We are now ready to construct the set of edges E:

$$E = \{(\{i\}, \{i+1\}, m) \mid m \in M \wedge s_i = m.tstamp\}$$

In other words, we order the messages chronologically, and create states between them. For consistency we use this system-state representation throughout the paper.

Relating the representations Let $G = (V, E)$ be a graph that interprets messages as states. Then we can construct a graph $G' = (V', E')$ that interprets messages as transitions and allows the same traces. Let $V = \{s_1, ..., s_n\}$ and r be a relation according to which we will perform the translation. Then G' can be constructed as follows:

$$\begin{aligned} V' &= \{0, ..., n\} \\ E' &= \{(i, j, s_j) \mid (s_i, s_j, r) \in E\} \cup \\ &\quad \{(0, i, s_i) \mid \neg\exists j : (s_j, s_i) \in E\} \end{aligned}$$



(a) Relational representation

(b) System state representation

Figure 4: An example to illustrate automated conversion of a relational representation in **(a)** to the system state representation in **(b)**.

As an example of this conversion, consider Figure 4. In graph **(a)**, G is represented in the relational form, while in **(b)** the same graph is represented in terms of system states.

We are also fairly confident that the reverse translation is also possible, at least for the chosen translation relation. It is also apparent that interpreting messages as states yields a more general representation, as it allows for a variety of different relations between messages.

4.2.3 FSM Representations Exploration

To explore the space of potential FSM representations Synoptic uses two dual operations: refinement and coarsening. These are illustrated in Figure 5. A variety of decisions that the graph refinement and coarsening algorithms have to make are encapsulated in the decision engine (explained in section 5). Here we overview the intuition behind refinement and coarsening.

Graph Refinement Graph refinement begins by grouping all messages of a particular type into a single state. The algorithm then starts to find counter-examples to this grouping, and refines the states accordingly. State refinement is based on a property that splitting the messages associated with a state into two sets preserves the temporal properties of the messages observed in the original trace.

State refinement allows decisions to be made depending on the properties of the entire partition. For example, if only a small number of states can be split out, the algorithm can decide not to split so as to maintain a more compact representation. In general, statistical measures can be used to drive the refinement (see discussion in section 9).

If certain refinements should be guaranteed, the initial graph can start with a more fine grained initial assignment of messages to states. At the moment we make

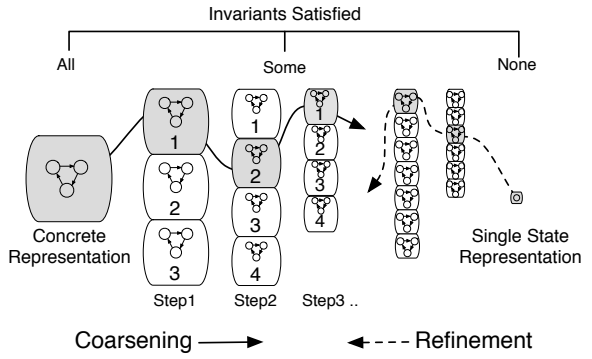


Figure 5: Coarsening and Refinement are dual operations on the graph representation. Squashed rectangles denote intermediate FSM representation, and the size of the squashed rectangle denotes representation size (e.g. number of states and transitions in the FSM graph). The most compact representation is a single state representation, while the least compact representation is the concrete representation. Refinement starts with a single state representation and can proceed (to the left) until the concrete representation is achieved. Coarsening starts with the concrete representation and can proceed (to the right) until a single state representation is achieved. Three steps of coarsening are labeled; the intermediate representations at each step are shaded squashed rectangles. At the top, the representations are related to trace invariant satisfaction. All invariants are satisfied in the concrete representation, and fewer are satisfied until a single state representation is reached, at which point no invariants are satisfied.

the initial assignment based on the user-supplied message type predicates.

Graph Coarsening Graph coarsening is the dual operation of graph refinement. The initial assumption is that all messages are different. When similarities are detected between some set of states, the states are merged into a single state while preserving the transitions of the initial states. A variety of state similarity notions may be used – behavioral, structural, or both. At the moment we only consider behavioral properties to decide whether to merge two states or not.

Graph coarsening takes the message type predicates into account and never merges states with different types. In this way, we achieve the same restriction as graph refinement.

5 Decision Engine

A variety of decisions that the graph refinement and coarsening algorithms have to make are encapsulated in the decision engine. For instance, the decision engine decides when the refinement or the coarsening process is considered complete. These decisions are encapsulated in the interface exposed by the decision engine to the algorithms. We now overview the three methods comprising this interface and the policies supported by each of these methods.

Invariant	**LTL$^-$ formula**	**Type**
x AlwaysFollowedBy y	$\Box(x \rightarrow \Diamond y)$	liveness
y AlwaysPrecededBy x	$\Box(y \rightarrow \Diamond^- x)$	safety
y NeverAfter x	$\Box(x \rightarrow \Box\neg y)$	safety

Table 2: Three types of detected invariants with corresponding LTL$^-$ formula and classification. These invariants are required to be never violated in the course of coarsening or refinement. In LTL, $\Box$ and $\Diamond$ mean *for all time*, and *eventually* respectively.

5.1 *checkGraphValidity*

The first important decision is whether the graph is considered a valid representation of the input traces. For this purpose the decision engine provides the method *checkGraphValidity*, defined as

$$checkGraphValidity \ : \ Graph \rightarrow Boolean$$

The decision engine can use different metrics to judge whether a representation is valid or not. Our only heuristic at the moment is to make sure that the representation satisfies a set of temporal invariants mined from the original input trace.

5.1.1 Temporal Invariants Satisfaction

The temporal relations between messages are potentially related to their causal dependencies. Because of this we intuit that strict temporal invariants should be preserved in the final representation of the trace. As an example, in the two phase commit protocol, the message TXAbort is sent *because* a node sent Abort, thus TXAbort occurs *after* Abort.

Mining invariants. We mine the patterns listed in Table 2 from the original input traces with the temporal relation between messages extrapolated from the timestamps. These patterns are partially based on the specification patterns and the classification scheme formulated by Dwyer et. al. [12]. Our algorithm first constructs the temporal graph with nodes representing messages, and then constructs the transitive closure of this graph. The invariants in Table 2 can then be deduced by considering all the outgoing and incoming transitions of a node. As an example, Appendix A lists all the temporal invariants mined for the two-phase commit protocol.

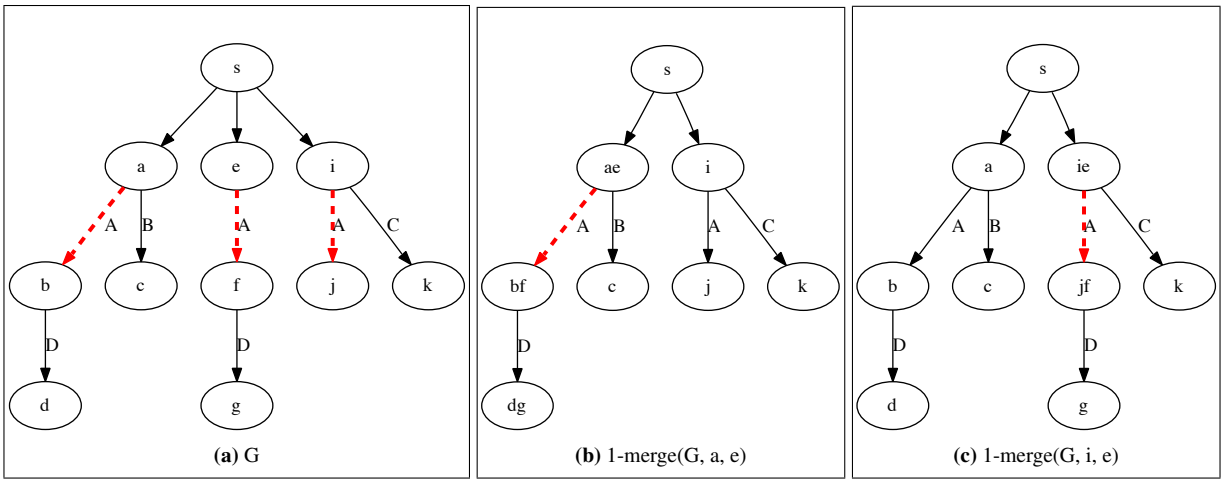


Figure 6: During merge, nodes may be merged in different orders, and order matters. That is, merging is non-associative. Consider the graph in **(a)** above. The result of executing 1-equiv(G) is {(a, e), (i, e)}. That is, node pairs (a, e) and (i, e) are 1-equiv. The dashed edges in **(a)** indicate this equivalence. Executing 1-merge(G, a, e) produces **(b)** above. In this graph node ae is not 1-equiv to node i. Executing 1-merge(G, i, e) produces **(c)** above. In this graph too, nodes a and ie are not 1-equiv. In **(b)** and **(c)**, bolded nodes indicate merged nodes, and the red dashed edge indicates the merged edges from **(a)**.

Checking an input graph. The mined invariants are checked against the input graph and True is returned only if the graph satisfies all of them.

5.2 *getMerge*

The second method exposed by the decision engine relates to graph coarsening. Intuitively, the purpose of this method is to break ties in the coarsening algorithm when there is a number of potential states to merge. The *getMerge* method takes as input a graph and a set of subsets of the nodes in the graph that are potential candidates for merging (i.e. which the algorithm is capable of merging). This method returns the element from the input set of subsets, which is the candidate the coarsening algorithm should merge. However, this method does not guarantee that the graph G' obtained after merging will satisfy $checkGraphValidity(G')$. Validity may not be satisfied because this method has no information about the transformation that will be performed on the graph by the coarsening algorithm.

$$getMerge \ : \ Graph \times \mathcal{P}(\mathcal{P}(S)) \rightarrow \mathcal{P}(S)$$

This method takes a set of possible merges as argument, and returns the best possible merge candidate set.

5.2.1 Choose to-merge partition at random

Our current policy to decide which nodes to merge is to pick a node at random. For GK-Tail, this results in the behavior specified in the original paper [22]. However, the shape of the resulting graph depends on the random choice, and the algorithm is thus non-deterministic. To see this, consider Figure 6. In graph **(a)**, node e is 1-equiv to nodes a and i, where 1-equiv is defined in section 6.1. Merging nodes e and a results in graph **(b)**, while merging e and i results in graph **(c)**. Graphs **(b)** and **(c)** are topologically different, and nodes ae and i in **(b)** as well as nodes ie and a in **(a)** are not 1-equiv. Moreover graph **(b)** is a more compact representation than graph **(c)**. The choice of which nodes to merge, therefore, has important implications – a non-deterministic choices is not guaranteed to find the global minimum, and the choice is not symmetrical. These properties can be directly attributed to the fact that k-quiv is not an equivalence relation, since it is based on subsumption, and not equivalence, of graphs.

5.3 *getSplit* Policies

The final method relates to how the graph can be refined. The *getSplit* method takes as input the graph and outputs a node and a predicate to be applied on the node that determines the split. This method can only be applied if there is at least some node that can be partitioned further. getSplit is defined as

$$getSplit \ : \ Graph \rightarrow Predicate \times S$$

This method takes a graph as argument, and returns a node, and a predicate. The predicate partitions the nodes in two sets, and this is the intended split. There are several possibilities for deciding on both, the node and the predicate to use for the split.

```
Input: (V,E)
Let S := k−equiv(G)
While S ≠ ∅
  Let (s1, s2) = DE.getMerge((V,E), S)
  Let (V',E') = k−merge((V,E), s1, s2)
  If DE.checkGraphValidity((G',V'))
    (V,E) := (V',E')
    S := k−equiv((V,E))
  Else
    S := S − {(s1,s2)}
Output: (V,E)
```

Figure 7: The GK-Tail algorithm

5.3.1 Counter-example guided

An informed method to select the state to split is to use counter-example to the invariants that were initially mined. For every unsatisfied invariant, there is a violating trace, and the node to split can be chosen from such a trace. There are several possible ways to choose the node. In Bikon we pick the first violating node from the trace. We do so because it provides us with an intuitive measure of progress – splitting the first violating node leads us to hope that the resulting paths have no violations, or have at least one fewer violating nodes. The predicate returned by getSplit is the transition in the invalid path.

6 The GK-Tail Algorithm

We now describe our adaptation of the GK-Tail algorithm. This algorithm consists of two steps. First the algorithm merges all input-equivalent traces, which are sequences of the same messages but with potentially different fields. This is an initial data-reduction step. Second, the algorithm starts with the most refined graph representation of the trace and proceeds to coarsen this graph at each step by merging some number of nodes. It terminates when it cannot find any nodes to merge, either because they are no two nodes that are similar enough, or because a merge of any two nodes would violate a temporal trace invariant.

6.1 Coarsening

Let $G = (S, M)$ be the graph obtained by treating every trace as a sequence of states with edges between the states representing consecutive messages in the trace. We then define two helper functions:

- `k-equiv(S,M)` returns $T \subseteq S \times S$ such that for all $(s,t) \in T$ we have that s is k-similar to t in G. By this we mean that the directed graph rooted at s, which is a subgraph of G with maximum path length of k is a subgraph of a similarly constructed graph rooted at t. There is a stronger version of k-similarity which mandates that the subgraphs be equivalent, however, we ignore this form in this paper.
- `k-merge(`G, s_1, s_2`)` returns G' that is obtained from the graph G by merging the nodes s_1 and s_2 up to depth k.

Now we can formulate our modified GK-Tail algorithm as shown in Figure 7. The k in this algorithm is a key parameter that can be thought of as the degree of dependency between messages. If messages are completely independent, a k value of 1 is sufficient. However, if a message influences a message downstream that is n messages away, k should be set to a value $\geq n$. A smaller k value produces more compact representations, but it does so at the loss of potential dependency information. A larger value of k retains this information, however, it can be difficult to set the right value of k. Moreover, a static value of k might not be appropriate for all steps of a protocol. At the moment our algorithm uses a static value of k.

6.2 Complexity of GK-Tail

In the worst case, the GK-Tail algorithm will have to consider all pairs of states in the current graph, if they are all k-equivalent, before finding a pair that, when merged, gives a valid representation. When a merge is performed, exactly one state is removed from the graph. Thus, in the worst case the time-complexity of the algorithm is $O(kn^3)$. Predictably, this does not scale well and in practice does not handle input traces with more than 1000 entries.

6.3 Divide and Conquer Optimization

We address the poor scalability of GK-Tail algorithm by employing a divide-and-conquer approach. We make the observation that the GK-Tail algorithm may be applied independently to subgraphs of the original graph as long as the shortest path connecting the subgraphs is no shorter than k. We implement this by dividing the initial graph and performing GK-Tail on each of the resulting sub-graphs before merging the resulting graphs and performing GK-Tail on the merged graph. The theoretical worst-case time-complexity is unchanged, but we find that this divide and conquer approach works well in practice and makes the algorithm scale effectively to traces of more than 20,000 entries (see section 8.5 for more).

```
Input: (V,E)
(V,E) := partition(V,E)
While not DE.checkGraphValidity((V,E))
  Let (p, Δ) = DE.getSplit((V,E))
  Let Δ1 = {s ∈ Δ | ps}
  Let Δ2 = {s ∈ Δ | ¬ps}
  V := V ∪ {Δ1, Δ2} − {Δ}
  E := ex−abs(V)
Output: (V,E)
```

Figure 8: The Bikon algorithm

7 The Bikon Algorithm

We modified a partition refinement algorithm [23] to build a bisimulation-inspired minimization that we call *Bikon*. This algorithm is novel because of a key property – the algorithm creates FSMs that preserve some temporal properties of messages – e.g. certain message orderings.

We interpret messages as states as described earlier. In the beginning, we partition the graph by the data fields, i.e. we merge all messages with certain data-fields into one state. We assume there is routine `partition` available for this purpose.

We then refine the graph until the *checkGraphValidity* call to the decision engine returns true. If *checkGraphValidity* returns false, we let the decision engine choose a predicate p and a state Δ to split according to the predicate. After adding two new states Δ_1, Δ_2, we have to update the edges in the graph. For this we use existential abstraction, i.e. we insert edges between states if two messages from the states are in the corresponding relation.

$$\text{ex-abs}(V) = \{(s_1, s_2, r) \in V \times V \times R \mid \exists m_1 \in s_1 \exists m_2 \in s_2 : m_1\, r\, m_2\}$$

The pseudo-code for Bikon is shown in Figure 8.

8 Evaluation

We implemented Synoptic in Java. The core of the system (excluding tests) is implemented in approximately 3,800 lines of code. Internally, the GK-Tail and the Bikon algorithms use the system-state and the relational representations respectively. Internally, the decision engine uses the relational representation – GK-Tail graphs are converted every time it uses the decision engine. We use dot [1] to generate the visual representations of Synoptic output. As a side-note, for the non-synthetic systems we evaluate in this section, we additionally augment the edges in the system-state representations with a count of the number of messages observed in the trace along that transition.

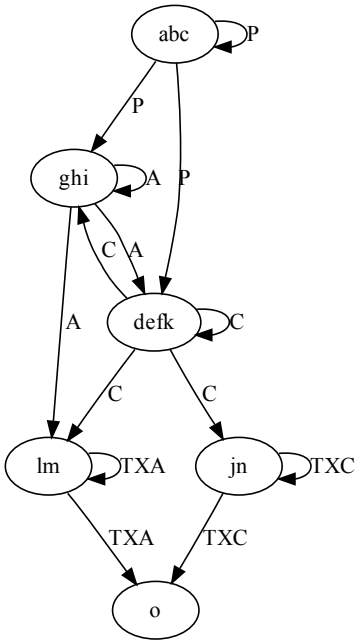


Figure 9: A representation derived for a synthetic two-phase commit trace using GK-Tail with k=1 and with temporal invariant checking disabled. States are labeled with letters whose combination indicates a particular merge of single letter states of the FSM in Figure 11.

In the remainder of this section we evaluate our work by applying Synoptic to a variety of distributed systems and protocols. We start off by using Synoptic to learn about the Twitter API and in this context consider the difficulty of modifying an existing system to log messages. We then elucidate some of the properties of our two algorithms by applying Synoptic to synthetic traces of the two-phase commit protocol. Next, we use Synoptic on traces of the Network File System to illustrate the importance of choosing appropriate message partitions. We then report on a user study with a system developer in which Synoptic was applied to traces of a system intended for findings the reverse traceroute between two internet hosts. We conclude this section by evaluating Synoptic's performance across a variety of dimensions.

8.1 Modifying Existing Systems for Logging

One of the questions we wanted to answer is how difficult is to instrument an existing system to output traces in a form that Synoptic can process automatically? To provide an initial answer to this question we designed a

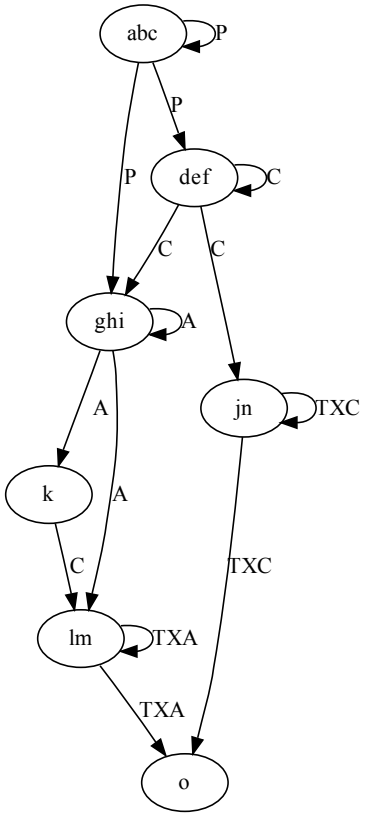


Figure 10: A representation derived for a synthetic two-phase commit trace using GK-Tail with k=1 and with temporal invariant checking enabled. States are labeled with letters whose combination indicates a particular merge of single letter states of the FSM in Figure 11.

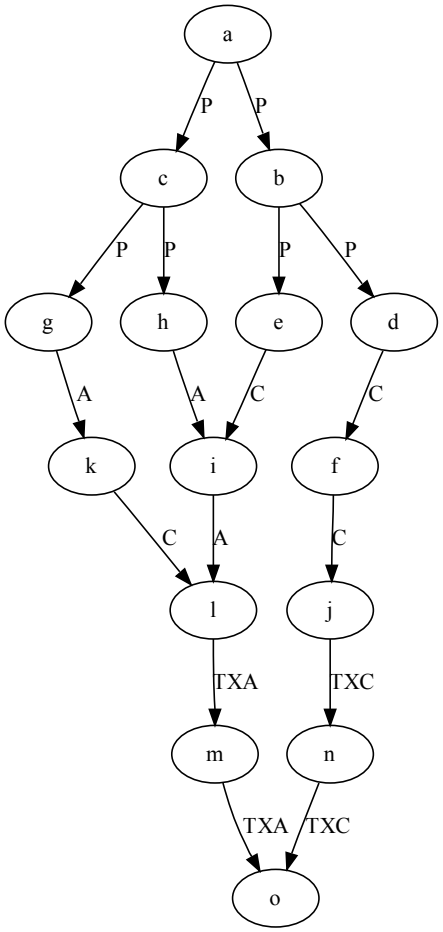


Figure 11: A representation derived for a synthetic two-phase commit trace using GK-Tail with k=2 and with temporal invariant checking enabled.

protobuf message specification interface [2] to Synoptic, and instrumented an existing system to use it.

8.1.1 Leveraging protobuf message specifications

Because manual system instrumentation is necessary, we decided to leverage protobuf [2] message specifications to simplify our task. This specification captures all the message fields and their types. The availability of a protobuf specification may enable future structural analysis of the messages observed, and simplifies serialization and deserialization of system messages to and from a trace file. These specifications also provide a natural way to distinguish messages according to their type.

We manually wrap all captured messages into a custom message type, the structure of which is depicted in Figure 12. This message specification augments captured messages with meta-information, such as the source and destination addresses, and a message timestamp. The procedure of formulating this specification must be manually repeated for each system that we study.

```
message SynopticMessage {
  required string src = 1;
  required string dst = 2;
  required int64 tstamp = 3;
  required string message_type = 4;
  required bytes system_message = 5;
    //system_message is a ByteStream
    //encoding of a protobuf format
    //message for the system in question
}
```

Figure 12: Protobuf format of a wrapped captured message.

We then combine the protobuf wrapper with a Tracer class which provides static methods for logging messages in a system. With these two frameworks in place, instrumentation of existing systems, especially those using protobuf formatted messages, is straightforward.

8.1.2 Twitter Java library evaluation

We apply Synoptic to traces generated by the Java wrapper of the Twitter API [3]. Modification of the Java Twitter API required us to add calls to Tracer.log() at two locations in the code where protobuf messages were about to be converted to an outgoing on-the-wire format. In addition we added this call to where incoming messages are processed. In total we added **three** lines of code to an existing system to automate the processing of its message logs with Synoptic.

We built a basic Twitter client that successfully performed a range of Twitter operations using the instrumented Twitter API, logging all the messages sent and received from the Twitter server. These messages were then processed by Synoptic automatically. Figure 16 (in the Appendix) illustrates the resulting collection of FSMs generated by Bikon (and then converted into system-state representation). The GK-Tail algorithm yielded a similar representation, except that the final states of all the FSMs were merged into a single state at the very end. We think the resulting representation is an intuitive means of documenting the Twitter API. Though the original system is not complex, this example illustrates that it is straightforward to modify an existing systems to log messages for automated analysis with Synoptic.

8.2 Two-Phase Commit Protocol

We evaluated our framework on synthetic traces of a three-node two-phase commit protocol (with one node acting as the transaction manager). We used a synthetic trace to understand the impact of our choices on the design of GK-Tail and Bikon algorithms.

8.2.1 GK-Tail

The graph generated with GK-Tail with $k = 1$ and with disabled invariant detection is depicted in Figure 9. Considering that this representation can generate an invalid trace containing both, (A)bort and (TX-C)ommit, it is clear that this is not a good representation of the traces. The representation also contains self-loops, and thus allows for behavior that is valid but which was not observed in the input traces.

The same input was also processed with $k = 1$ and with invariant detection enabled and is depicted in Figure 10. The spurious trace is gone, since the crucial invariant was enforced by the decision engine. However, the FSM contains self-loops, and thus allows for an unbounded number of (A)borts and (C)ommits to occur before the transaction is eventually finished. This does not capture the important property that only two nodes participate in the protocol.

The most concise representation of the graph is obtained with $k = 2$ and is shown in Figure 11. In this case, the invariant detection does *not* alter the output. This is because the temporal invariant span is two, and $k = 2$ guarantees that traces up to this length are preserved. The significant improvement over Figure 10 is the absence of self-loops. Although the graph clearly shows that no more than two commits can occur, it misses the opportunity to merge states g, h, e, and d.

Although checking temporal invariants does not alter the final representation for $k = 2$, it is important to note that choosing the right value of k is difficult without deep understanding of the system. Additionally, our results indicate that the choice of k is crucial even in the presence of invariants. On the other hand we have shown that small values of k in combination with invariant detection lead to smaller and - more importantly - correct, graph representation for our two-phase commit traces.

8.2.2 Bikon

Bikon is able to produce the trace in Figure 10 as well. The trace in Figure 11 is out of reach, and we experienced the same problems with self-loops. We feel that the strength of Bikon lies in its ability to consider statistics, i.e. whether a node should be split could potentially made dependent on the properties of the other nodes in that partition. This is a direction we intend to pursue in our future work.

8.3 Network File System

We applied Synoptic to a set of Network File System collected by the Self Organizing Storage project [13] and evaluated the resulting representations. The contribution of this evaluation was two-fold. This set of real-world traces exposed the limitations of our framework as well as demonstrated the importance of choosing an intelligent partitioning strategy.

Initially, we applied our framework to NFS traces with no partitioning. NFS [4] is a stateless protocol (see section 1.6 in [4]) which implies that a large enough input trace without any partitioning results in a representation that is a fully connected graph. This happens because in a stateless protocol it is possible for any message to follow any other message. Thus, in the absence of temporal invariants in the trace, this demonstrates a fundamental limitation of our summarization approach.

We then considered the same traces partitioned by accessed file ID. This produces a set of traces with each trace representing successive NFS operations performed against a single file. While NFS is a stateless protocol, in practice interaction with a particular file is *not* stateless. The representation produced by Synoptic clearly

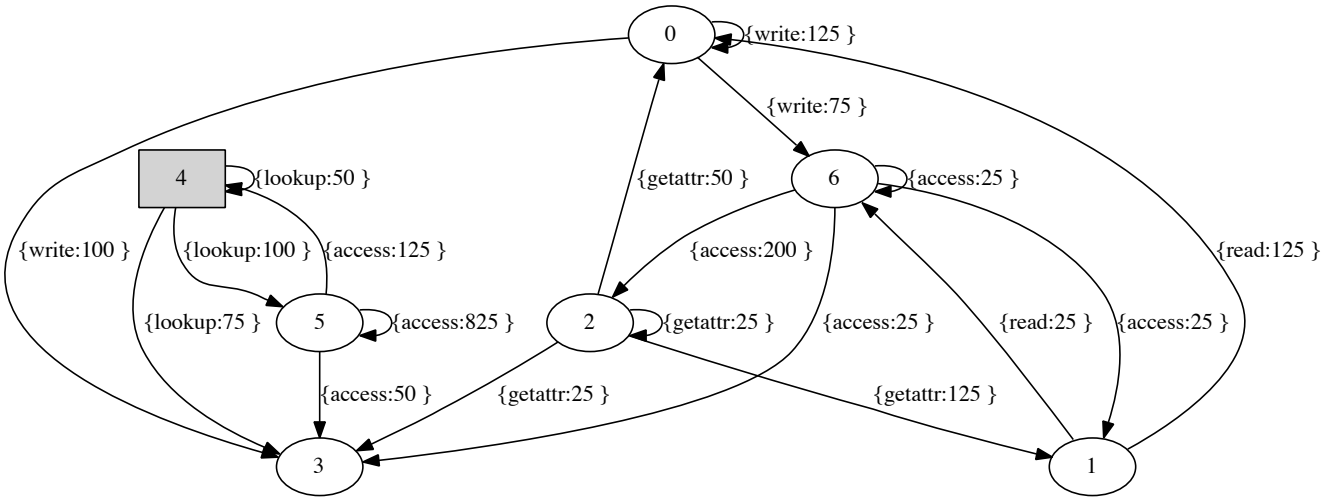


Figure 13: A representation derived for 2K lines of NFS traces partitioned by file ID. The rectangular grey node indicates the start state. Generated using GK-Tail with k = 1.

shows this, as seen in Figure 13. We believe that this representation may help to answer a variety of questions concerning file access patterns in NFS (e.g. which files are only read, only written, or a more complex combination thereof.)

8.4 User Study

We carried out a think-aloud user study with a distributed systems developer who built a system that determines the likely reverse traceroute from an arbitrary destination on the internet to a source host [5]. This tool relies on a distributed set of internet vantage points hosted by PlanetLab [6], and uses a variety of methods to find each segment of the reverse route. For example, these methods include using IP record route, and timestamp options [7, 8], and relying on IP spoofing from PlanetLab hosts [9].

For our study, we first met with the developer to receive a brief primer on how to interpret the system generated log files. We then developed a small parser (in approximately 100 lines of Python code) to extract semantically useful information from these log files by repeatedly matching a list of custom regular expressions. The output of this parser was then partitioned into traces such that each dealt with determining exactly one segment on the reverse path. The partitioned traces were then fed into Synoptic.

We then met with the developer once more and presented six Synoptic-generated representations. One of these traces, generated with GK-Tail, is shown in Figure 17 (in the Appendix).

8.4.1 User observations

The following are some user observations that we thought were particularly insightful. For each observation, we comment on its relevance and potential implications for Synoptic.

- In our study of the FSMs we noticed a few "spurious endpoints" in which path segment computation terminated in a non-terminating state. When these spurious endpoints were pointed out to the user, he explained that this was indeed possible. The computation may not terminate in one of the expected states because (a) the process was killed before completing, or (b) the forward traceroute failed, so the reverse traceroute was not attempted. The representation prompted an interesting question about system design.

- By looking at the representation, the user was not immediately able to tell whether an edge "initializing RR VP" (where VP stands for vantage point) could ever bypass "connecting to VP." Then after considering the matter some more determined that a connection to a vantage point is not re-established if its already present from a previous iteration. The representation prompted the user to remember an implementation detail.

- The user found the Bikon representation, in which messages are represented as states to be "pretty close" to his mental model of the system. The reason for this was that logged messages were used by the system to signify a particular system state. The user did not know how to interpret states in the GK-

Tail representation because they lack meaningful labels (they are randomly generated).

- Although the user favored the Bikon representation, he found it confusing to see multiple nodes with the same label. He thought that the partitioning made it difficult to reason about the graph, and that it enlarged the graph without adding much value. Large relational representations seem to be more challenging to interpret than the equivalent system-state representations.

- The user was able to point out distinct regions of the FSM where the system employed a particular method to determine a segment on the reverse path. Figure 17 illustrates the regions pointed out by the user overlayed onto the FSM as dashed enclosures with labels **(a)** - **(e)**.

- The user mentioned twice that it would be useful to have the ability to dynamically collapse and expand the representation to see detailed information about a particular branch in the FSM. He seemed to want more control over the layout, and was dissatisfied with `dot`'s inability to preserve layout between runs over similar data. Also, the user expressed interest in finding out which traces caused special transitions, especially in the context of potentially spurious traces.

- The user mentioned that the Synoptic representation can make it easy to discover unexpected paths in the system. He also found it useful to see how often certain paths were taken by the system.

- The user asked whether he could find out how long a loop can be taken before a certain output becomes impossible. This indicates that it might be useful to add labels to loops indicating the minimum and the maximum number of observed transitions.

- The user mentioned that he will be working on a patent application and that this kind of tool would come in useful in documenting the features of the algorithm and for including a diagram in the patent application.

Although this was a micro-study of a single application of Synoptic with a single user, we think that it demonstrates that Synoptic is useful. The user found the tool helpful for investigating complex system behavior. Additionally, based on our personal observations we believe that Synoptic can reduce the amount of time a person needs to spend to understand the various protocols making up a distributed system.

Var	Meaning
M	The total number of messages in the input trace
n	Number of partitions for the trace
r	Number of unique messages in type 1 traces
k	The GK-Tail k parameter (GK-Tail only)

Table 3: Notation used in the performance evaluation.

8.5 Performance Evaluation

In this section we benchmark the performance of GK-Tail and Bikon across a variety of dimensions. We measure an algorithm's performance in the total milliseconds the algorithm takes to complete its analysis. This number does not include the time it takes to read the input trace file from disk nor the time to output the final representation. All benchmarks were run on an x86 Windows 7 machine with an Intel Core 2 Duo (2.2 GHz) and with 2GB RAM. For all traces we used a sequence of $r = 10$ messages (i.e. $[m1, m2, m3, \ldots m10]$) that repeats a set number of times. Table 3 overview the variables we use as shorthand for our performance evaluation. To derive each data point in the following graphs we ran the algorithm three times and took the average of the three running times.

8.5.1 Impact of Optimization and Invariants on GK-Tail

Figure 14 plots the performance of trivial (unoptimized) GK-Tail, scalable (optimized) GK-Tail, and scalable GK-Tail with Invariants versus the number of partitions of the trace (in graph **(a)**) and total messages in the trace (in graph **(b)**). Scalable outperforms Trivial in all cases, with or without invariants. This indicates that the GK-Tail optimization significantly speeds up execution time. Invariants introduce a significant slow-down to scalable and trivial GK-Tail (trivial with invariants is not shown). In graph **(a)** including invariant inference with n < 4 partitions led to a timeout; likewise for M > 500 in graph **(b)**.

8.5.2 Bikon Performance

The Bikon algorithm is much more scalable than both versions of GK-Tail. Figure 15 shows the performance of Bikon without invariant inference versus the number of partitions (in graph **(a)**), and versus the number of total messages (in graph **(b)**). These graphs demonstrate that Bikon is very efficient. We do not show Bikon with invariants because the invariant inference algorithm is the same for GK-Tail and for Bikon, therefore Bikon experiences a slowdown similar to GK-Tail when invariants inference is enabled.

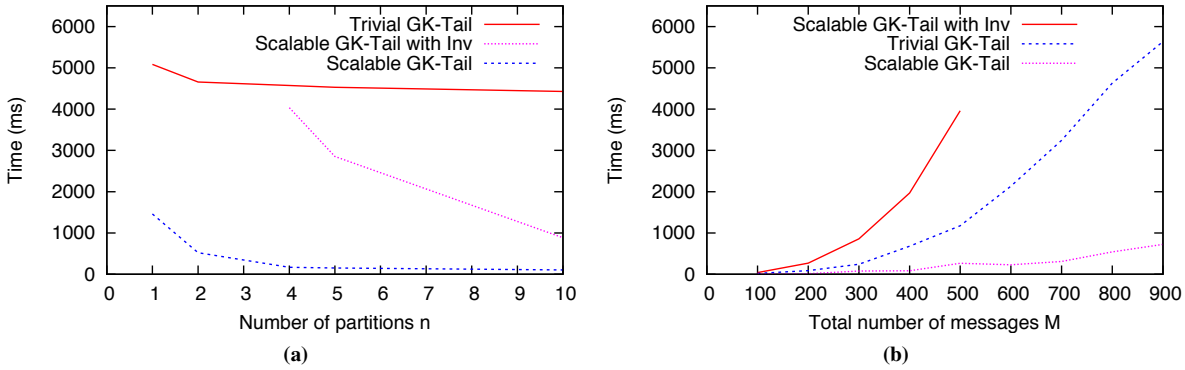


Figure 14: (a) Trivial, Scalable, and Scalable with Invariants GK-Tail versions running time with M=800, r=10, k=1, and varying n. All algorithms benefit from a higher number of partitions. Scalable GK-Tail outperforms trivial GK-Tail. **(b)** Trivial, Scalable, and Scalable with Invariants GK-Tail versions running time with n=2, r=10, k=1, and varying M. All algorithms take longer to run on longer traces, but scalable GK-Tail scales much better than Trivial GK-Tail for larger traces. Invariants introduce a significant slowdown for large traces.

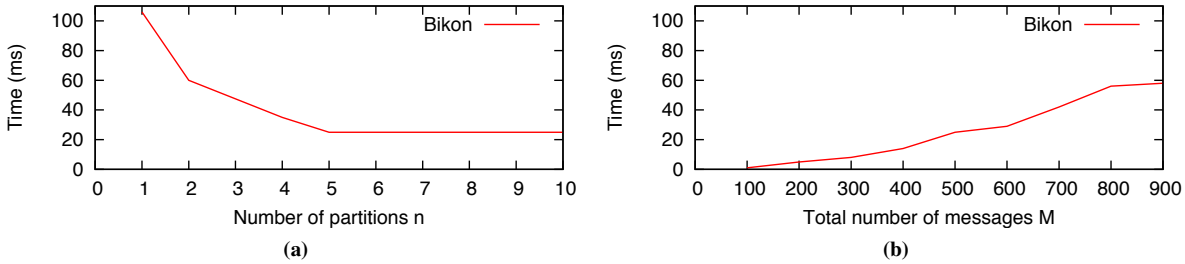


Figure 15: (a) Bikon running time with M=800, r=10, without invariant inference, and with varying n. **(b)** Bikon running time with n=2, r=10, without invariant inference, and with varying M. Both graphs indicate that Bikon is much more efficient than GK-Tail.

9 Discussion

Partitioning Strategies. Alternative partitioning strategies may yield different information about the trace. This is probably the most complicated task the user faces when using Synoptic. As an example of an alternative strategy consider partitioning messages based on the tuple of source and destination address pairs. This will produce representations that capture interactions between some two nodes in the system. We hope to explore such alternative strategies in our future work.

Message Capture. The protobuf library mentioned in section 8 diminishes the amount of work necessary to capture messages generated by a distributed system. However, there are other methods of achieving the same result. For example, all traffic on a local machine could be sniffed (e.g. using `tcpdump`), or the system could be modified to use a custom network library that logs all messages sent and received to persistent storage. However, in the case of lower-level capture methods such as tcpdump, more interpretation work must be performed to tease out message fields relevant to the analysis.

Alternative splitting policies. Synoptic uses temporal invariants on messages ordering to validate intermediate FSM representations. However, other types of invariants could be leveraged to guide the splitting decision of the Bikon algorithm. Structural invariants offer an alternative. For instance, Daikon [14] can be used to find a sensible splitting of a set of messages according to the data-fields associated with them. A split could be evaluated by cross-validating the messages from one candidate partition against the invariants from the other candidate partition, and vice versa. The more messages that do not satisfy the invariants, the better the split. Message fields over which structural invariants should be generated could be specified within the protobuf format.

Statistical methods. It may be possible to improve our framework with a suite of statistical methods. For example, it might be necessary to ignore parts of the trace to come up with a concise representation. This may correspond well to accounting for failure or other

unexpected events in the representation of the system. It is unclear, however, what specific statistical models or machine learning approaches are best to use since prior work on detecting such events and partitioning traces by protocol indicates that this is a challenging problem [10].

Further applications. To evaluate the system in [22], the authors use a test coverage metric. By ensuring that the set of tests covers every transition arc in the FSM representation, the tests can be considered to have a high degree of coverage. Our work may also be evaluated using this metric. Specifically, if we are able to trigger events in the system being studied, then we can drive execution towards those transitions and states that were observed infrequently.

10 Related Work

GK-Tail. The work by Lorenzoli, et. al. [22] is most closely related to ours, and inspired our use of the GK-Tail algorithm. In this work, multiple interaction traces are used to automatically generate an Extended Finite State Machine (EFSM). Each trace consists of a sequence of method invocations annotated with values for parameters and variables, and is generated by a single execution run of the program being analyzed. The resulting transition arcs of the EFSM are annotated with the relevant constraints on the data to transition from one state to the next state, providing a behavioral view of the program that includes how the input data affects the behavior. This algorithm is beneficial in understanding program behavior and in generating tests with better coverage. Our work generalizes GK-Tail to consider multiple k-future equivalent nodes, adapts the algorithm to the domain of distributed systems, and further modifies it to respect temporal constraints between nodes.

Another closely related work by Krka et. al. [18] addresses FSA generation for objects, i.e. a state based representation of legal method call sequences. The approach first generates a invariant graph based on inferred invariants from Daikon. Then the method call sequences are compressed using $kTail$, but merges that would lead to invalid call sequences according to the first FSA are restricted. The result is less likely to contain invalid traces.

A third use of $kTail$ is found in a paper by Lo et. al. [21]. In this work as the execution traces are mined for temporal properties which are then used to steer the $kTail$ algorithm to ensure that a given merge will not produce a graph that violates any of the temporal constraints. The paper focuses on the methods used to mine temporal properties and on the complexity of enforcing those constraints during the $kTail$ algorithm. It shows that $kTail$ with steering produces graphs with much higher precision. This methodology is similar to the "steering" of GK-Tail provided by the decision engine in Synoptic. The paper's main focus is the difference between $kTail$ with steering and without steering, while Synoptic incorporates these methods into a more comprehensive distributed systems analysis tool.

Bisimulation. Bisimulations are simulation relations that provide a strong notion of similarity for relational structures. They emerged in different fields [24], and their key property is preserving certain properties of the relational structure, for example, two strongly bisimilar transition systems are guaranteed to satisfy the same set of LTL formulas. Building on this property, an important application in model checking is model minimization [15]. Our Bikon algorithm is a modification of a partition refinement algorithm [23]. Bikon uses invariants to determine which state to split next, and we stop splitting much earlier (once all invariants are satisfied), which results in a coarser representation.

Debugging distributed systems. Distributed systems are notoriously difficult to get right. Recent efforts by the systems community target bug finding in distributed systems with model checking. MODIST [26] explores the inter-leavings of concurrent events in an unmodified distributed system, thereby model checking the live system. CrystallBall [25] explores the state space of an actively executing distributed system, and when inconsistencies in possible future states are found, CrystallBall can be used to steer the distributed system away from buggy states. Magpie [11] and X-Trace [16] operate a fine granularity of request processing in a distributed system. Magpie is primarily used for performance debugging and X-Trace is used to understand pipelined request processing. All these tools, however, do not target the extraction of system properties that may be used to understand the behavior of the system they check.

Property representations produced by Synoptic may be leveraged by these tools to guide model checking, and Synoptic may use these tools to target system execution towards states for which Synoptic lacks information to derive a concise representation.

Inferring temporal properties of programs. Perracotta [28] is a tool to mine temporal properties from program event traces. It has been used to study traces of function calls for program evolution [27]. It has also been made scalable and robust to analysis of imprecise event traces to understand behavior and uncover bugs in very large code-bases such as the Windows Kernel and the JBoss application server [29]. Perracotta first instruments the program to output event information – all prior work uses method entrance and exit points. Second, the instrumented program is run – Perracotta relies on exten-

sive test suites and random exploration of method calls to generate a broad range of event traces. Finally, the generated traces are used to infer program properties. The system generates candidate temporal patterns and then attempts to gain evidence that indicate the pattern holds by scanning through the trace. Patterns are expressed in terms of quantified regular expressions which are similar to regular expressions. The system uses a partial order hierarchy of properties that are built on the response pattern – the simple cause effect relationship between some two events.

Unlike Synoptic, Perracotta considers a totally ordered trace of events and does not consider properties that might be of interest in the domain of distributed systems. For instance, events may be concurrent and cannot be ordered with respect to one another. Additionally, Perracotta does not make use of any relationships between event data (e.g. method arguments/message payload).

Hidden Markov Models. A classic approach to modeling a system with unobserved state but with observations that may imply certain system state and state transitions are Hidden Markov Models (HMMs). The HMM model is powerful but it relies on knowing the set of states, as well as the transition and emission probabilities for transitioning between hidden states and emitting an observation in a state respectively. In our setting, the system state is not observable so the HMM model is applicable. However, from message traces alone it is not even clear what is the right number of system states to use for the HMM model. As well, the transition and emission probabilities are not known. Using HMMs in our setting is therefore impractical because of the large number of inputs that must be additionally supplied by the user.

11 Conclusion

In this paper we presented Synoptic, a tool to summarize message activity in a distributed system and to output a finite state automata representation of this activity. Synoptic supports two algorithms – Bikon and GK-Tail, which refine and coarsen the graph representation respectively. We presented the technical details behind these algorithms, and the ways in which Synoptic abstracts away the policy to guide these algorithms into what we term the decision engine. Our results indicate that Synoptic produces representations for real distributed systems traces, which can aid in understanding complex system behaviors. In addition, we carried out a user study which suggests that Synoptic-produced representations can verify the intuition of a distributed systems developer. Synoptic is an open source tool, which can be found at the following URL: `http://code.google.com/p/cse503-system-analysis/`

Acknowledgments

Invaluable feedback on earlier drafts of this report by Michael Ernst were instrumental in shaping our presentation and ideas into their present form. We also thank him for making copious time to meet with us on demand throughout the quarter. We also gratefully acknowledge Ethan Katz-Bassett for his time to explain his reverse traceroute system and for his participation in our user study.

References

[1] Graphviz - Graph Visualization Software, http://www.graphviz.org/. Accessed March 8, 2010.

[2] Protocol Buffers - Google's data interchange format, http://code.google.com/p/protobuf/. Accessed January 13, 2010.

[3] A Java wrapper around the Twitter API, http://code.google.com/p/java-twitter/. Accessed March 8, 2010.

[4] NFS Version 3 Protocol Specification, http://www.ietf.org/rfc/rfc1813.txt. Accessed March 7, 2010.

[5] Reverse Traceroute, http://www.cs.washington.edu/research/networking/astronomy/reverse-traceroute.html. Accessed March 9, 2010.

[6] PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services, https://www.planet-lab.org. Accessed March 9, 2010.

[7] IPv4 Specification, Record Route option. http://www.ietf.org/rfc/rfc791.txt. Pg. 20, 21. Accessed March 9, 2010.

[8] IPv4 Specification, Timestamp option. http://www.ietf.org/rfc/rfc791.txt. Pg. 22, 23. Accessed March 9, 2010.

[9] IP address spoofing, http://en.wikipedia.org/wiki/IP_address_spoofing. Accessed March 9, 2010.

[10] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma. Constellation: automated discovery of service and host dependencies in networked systems, no. msr-tr-2008-67, april 2008.

[11] J. Domingue and M. Dzbor. Magpie: supporting browsing and navigation on the semantic web. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 191–197, New York, NY, USA, 2004. ACM.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[13] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive nfs tracing of email and research workloads. In *FAST '03: Proceedings of the Second Annual USENIX File and Storage Technologies Conference*, pages 203–216, San Francisco, CA USA, 2003. USENIX Association.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[15] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[17] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[18] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the New Ideas and Emerging Results Track at the 32nd International Conference on Software Engineering (ICSE10)*, 2010.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[20] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[21] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 345–354, New York, NY, USA, 2009. ACM.

[22] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[23] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[24] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):1–41, 2009.

[25] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[26] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: transparent model checking of unmodified distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[27] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 340–351, Washington, DC, USA, 2004. IEEE Computer Society.

[28] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 23–28, New York, NY, USA, 2004. ACM.

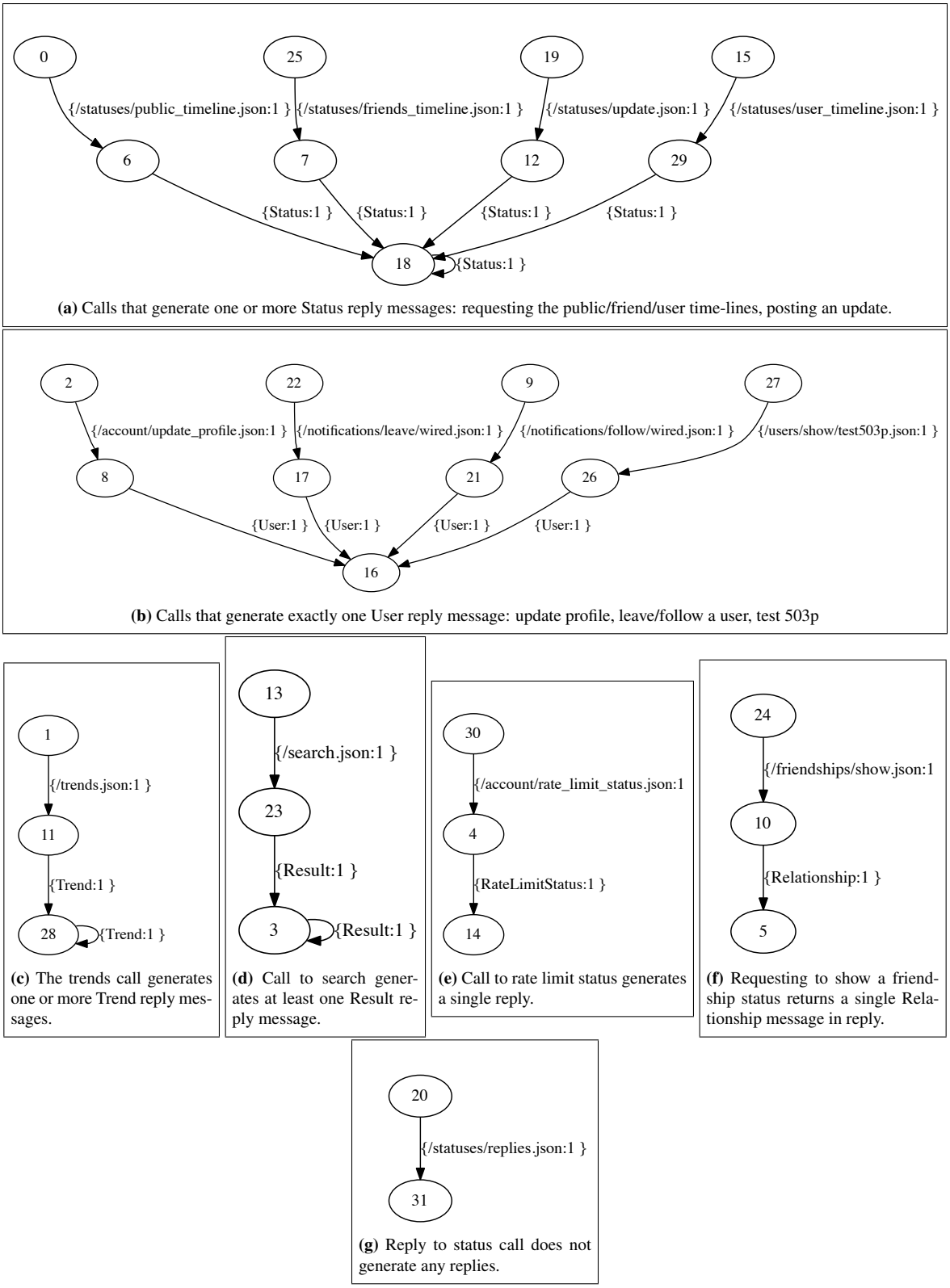
[29] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM.

x AlwaysPrecedes y	x NeverFollowedBy y	x AlwaysFollowedBy y
P, P	C, P	A, TXA
P, C	A, P	
P, A	A, TXC	
P, TXC	TXC, TXA	
P, TXA	TXC, P	
C, TXC	TXC, C	
A, TXA	TXC, A	
	TXA, C	
	TXA, P	
	TXA, TXC	
	TXA, A	

Table 4: Temporal invariants mined for the two-phase commit protocol.

A Temporal Invariants Mined for Two-Phase Commit Protocol

Table 11 lists all the temporal invariants mined from valid two-phase commit protocol traces.



(a) Calls that generate one or more Status reply messages: requesting the public/friend/user time-lines, posting an update.

(b) Calls that generate exactly one User reply message: update profile, leave/follow a user, test 503p

(c) The trends call generates one or more Trend reply messages.

(d) Call to search generates at least one Result reply message.

(e) Call to rate limit status generates a single reply.

(f) Requesting to show a friendship status returns a single Relationship message in reply.

(g) Reply to status call does not generate any replies.

Figure 16: The collections of state machines **(a)**-**(g)** depict the various Twitter client API message generating calls and respective server replies. Each state machine is a single interactions with the server. Generated using Bikon.

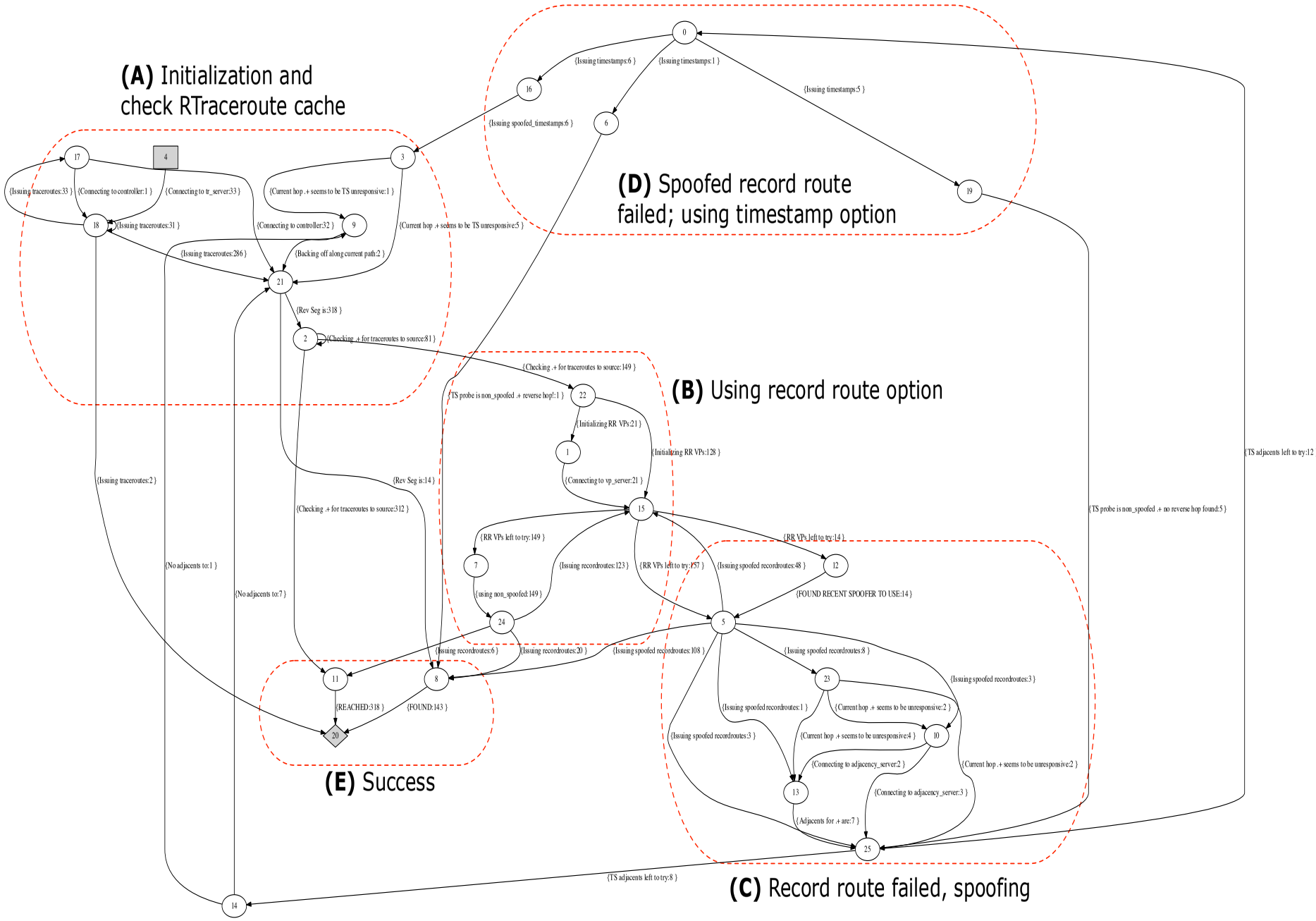


Figure 17: The state machine for the reverse traceroute system. Each partition is a computation of a single segment on the reverse path. The grey rectangle is the start state, and the grey diamond is the terminating state. Generated using GK-Tail. The overlays **(a)** - **(e)** (specified with red dashed lines) were generated by the developer during our user study, and correspond to state machines for each type of method the system uses to determine a single segment on the reverse path.