Causeway: A message-oriented distributed debugger

Terry Stanley, Tyler Close, Mark S. Miller

HP Laboratories
HPL-2009-78

Keyword(s):
event loop, distributed, debugging

Abstract:
An increasing number of developers face the difficult task of debugging distributed asynchronous programs. This trend has outpaced the development of adequate debugging tools and currently, the best option for many is an ad hoc patchwork of sequential tools and printf debugging. This paper presents Causeway, a postmortem distributed debugger that demonstrates a novel approach to understanding the behavior of a distributed program. Our message-oriented approach borrows an effective strategy from sequential debugging: To find the source of unintended side-effects, start with the chain of expressed intentions. We show how Causeway's integrated views - describing both distributed and sequential computation - help users navigate causal pathways as they pursue suspicions. We highlight Causeway's innovative features which include adaptive, customizable event abstraction mechanisms and graphical views that follow message flow across process and machine boundaries.

External Posting Date: April 21, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: April 21, 2009 [Fulltext]

© Copyright 2009 Hewlett-Packard Development Company, L.P.

Causeway: A message-oriented distributed debugger

Terry Stanley
Teleometry Design
tstanley@teleometry.com

Tyler Close
Hewlett Packard Labs
tyler.close@hp.com

Mark S. Miller
Google Research
erights@google.com

Abstract

An increasing number of developers face the difficult task of debugging distributed asynchronous programs. This trend has outpaced the development of adequate debugging tools and currently, the best option for many is an ad hoc patchwork of sequential tools and `printf` debugging.

This paper presents Causeway, a postmortem distributed debugger that demonstrates a novel approach to understanding the behavior of a distributed program. Our *message-oriented* approach borrows an effective strategy from sequential debugging: To find the source of unintended side-effects, start with the chain of expressed intentions.

We show how Causeway's integrated views – describing both distributed and sequential computation – help users navigate causal pathways as they pursue suspicions. We highlight Causeway's innovative features which include adaptive, customizable event abstraction mechanisms and graphical views that follow message flow across process and machine boundaries.

1. Introduction

The creative effort of designing and building software is a process of defining goals and implementing plans to satisfy the goals [1]. This is an iterative process applied at all levels, from the broad scope of a large-scale architecture to the narrow focus of a specialized algorithm. The resulting program expresses the authors' intentions. When actual program behavior is not what was intended, debugging is required.

The creative effort of debugging necessarily includes mapping observed behavior to a mental model of the original intentions to discover misconceptions as well as semantic and logic errors. Watching program execution at an appropriate level of detail and without interruption, supports this cognitive effort. Our development tools support this well in the case of sequential single-thread computation, but increasing numbers of developers write distributed asynchronous applications.

One conventional approach to distributed programming [27, 28, 29, 30] involves sequential processes sending and receiving messages on channels. In this case, when searching for the cause of a bug symptom, it is natural to look backward within the suspect process first, since each process is a single conventional program. This may be done using a conventional sequential debugger, a distributed debugger, or some combination. Prior distributed debuggers emphasize this process-oriented view of distributed computation [31].

With the emergence of the web as an application platform, *communicating event loops* are rapidly becoming the mainstream model for distributed computation. The web browser runs multiple isolated JavaScript programs. Each runs as an event loop, processing user interface events as well as asynchronous messages from a server [32]. With upcoming web standards, JavaScript event loops within the browser will be able to send asynchronous messages to each other [4, 5] and to multiple servers [6, 7].

In the communicating event loops model, there are no explicit receive operations; rather, each received message notifies a callback, spawning a sequential call-return computation which executes to completion. The stack becomes empty before the next incoming message is processed. For this style of distributed computation, the conventional debugger's stack view only reaches back to the last receive – the one that spawned the current stack. Since each received message is processed to completion separately, these computations are largely independent, so earlier state within the same process is likely less relevant than the process that sent the message.

We present Causeway, an open source postmortem distributed debugger for examining the behavior of distributed programs built as communicating event loops. Our *message-oriented* approach follows the flow of messages across process and machine boundaries. We discuss our experience with the Waterken web server [2] which we have instrumented to generate Causeway's language-neutral trace log format [3].

Other systems built on event loops include [8, 9, 10, 11]. Systems combining communicating event loops with promises include [12, 13, 14, 15, 16]. Causeway and Waterken also support promises, but promises are beyond the scope of this paper. See [17] for a comprehensive discussion of the promise-based approach.

Distributed programs often generate overwhelming amounts of trace data. The search for a bug must employ effective strategies for choosing what to pay attention to and what to ignore.

Noticing a bug symptom – a discrepancy between expectations and actual behavior – usually does not reveal the bug. The symptom is a consequence of the bug. Starting with a bug symptom, debugging proceeds by searching for an earlier discrepancy that accounts for this one. The search continues until a discrepancy caused directly by a logic error, rather than an earlier discrepancy, is discovered. This search does not proceed backward in time, but rather, backward in causal influence. To help guide the search, we need to navigate the structure of distributed causality.

Lamport's seminal paper [18] introduces the *happened-before* relation. This partial order models potential causality between events: If event *x* happened before event *y* it is possible that *x* influenced *y*. If *x* did not happen before *y*, then, to explain a discrepancy at *y*, we can safely ignore *x*. The partial order defined by happened-before thus helps narrow the search for a bug, but too many candidates remain. How can we narrow the search further? We found our first strategy in sequential debugging practice.

Sequential debuggers help us focus on the *intended causality* first, by following the chain of requests expressing intended computational effects. The stack view, central to sequential debugging, shows the nesting of calls expressing a chain of requests. Noticing the utility of the stack view for navigating call-return order, we realized that support for the distributed counterpart – message order – could be as useful for understanding distributed computation [19].

Message order helps us find the general neighborhood of the next discrepancy, but is often insufficient by itself. Process order and call-return order are then needed to explore that neighborhood in detail. Causeway's user interface supports navigation using multiple orders.

These event orders often contain repeated patterns of events – a mass of detail obscuring the clues of interest. Our next strategy, *event abstraction* [20], aggregates event patterns into abstract events. With Causeway's event *promotion*, *filtering*, and *aggregation*, users customize this abstraction according to which details they currently consider relevant.

Causeways primary innovations:

- Shifting emphasis from process order to message order.
- Integrated navigation across multiple orders.
- Event abstraction by customizing promotion, filtering, and aggregation.

The paper is organized as follows. Section 2 introduces the distinctions we use to discuss distributed causality. Section 3 presents the communicating event loops concurrency model. Section 4 presents our support for event abstraction. Section 5 shows how Causeway's user interface brings these elements together during the hunt for a bug. We conclude with a discussion of related work and our future plans for Causeway.

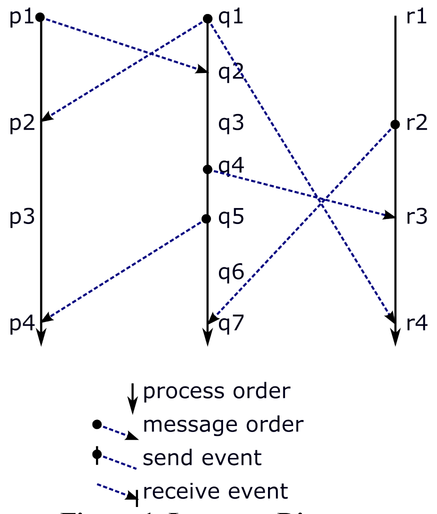


Figure 1. Lamport Diagram

2. Distributed Causality

Lamport defines the happened-before relation to model potential causality between events [18], which he explains in terms of space-time diagrams such as Figure 1. The vertical lines represent processes; each process consists of a sequence of events. Each event is either a local event or the sending or receiving of a message. Local events are loosely defined to be meaningful units of computation, chosen to best describe the behavior of the particular system being examined. The diagonal lines represent messages sent between processes.

Events within each process are fully ordered. Event p3 happened before p4 in *process order* because it occurred earlier in the same process, and so may have affected state p4 depends on. Event q5 happened before p4 in *message order* because q5 is the sending of the message that was received at p4. (In Actor terminology, process order is *arrival order;* message order is *activation order* [21]). The transitive closure of the union of these two orders defines the happened-before relation. A discrepancy observed at p4 may be caused by a bug at q1 but not r1.

Although introduced to illustrate a formal relation, Lamport diagrams, with a visual emphasis on process order, became the standard visualization of distributed computation [31]. (The time axis in Figure 1 is inverted from Lamport's original diagrams.)

Process order shows *state-based causality* – how a process reacts to later messages depends on how it changed state in response to earlier messages.

The happened-before relation tells us all the possible places that might have caused the bug, but gives us no guidance about where to look first.

2.1 A Distributed Asynchronous Example

Requests often have responses. Starting with one-way asynchronous messages, two patterns for handling

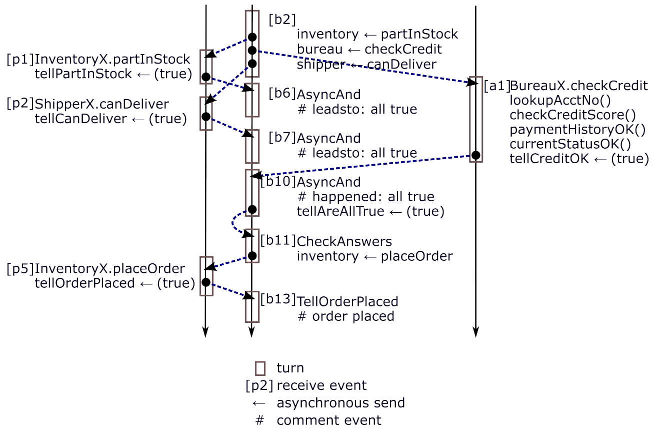
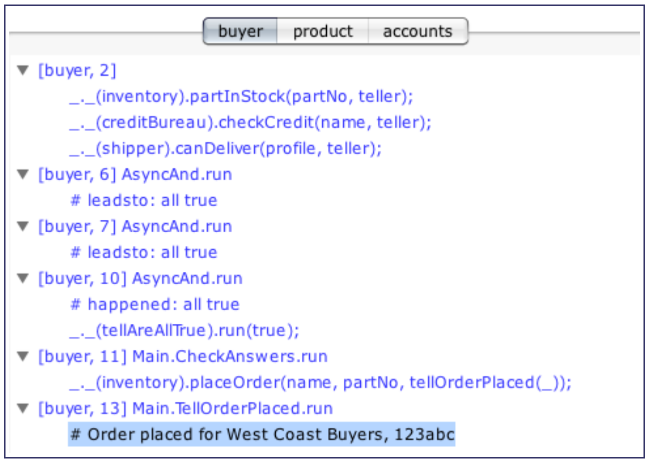


Figure 2. Process Order



```
▼ [buyer, 2]
      _._(inventory).partInStock(partNo, teller);
      _._(creditBureau).checkCredit(name, teller);
      _._(shipper).canDeliver(profile, teller);
▼ [buyer, 6] AsyncAnd.run
      # leadsto: all true
▼ [buyer, 7] AsyncAnd.run
      # leadsto: all true
▼ [buyer, 10] AsyncAnd.run
      # happened: all true
      _._(tellAreAllTrue).run(true);
▼ [buyer, 11] Main.CheckAnswers.run
      _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
▼ [buyer, 13] Main.TellOrderPlaced.run
      # Order placed for West Coast Buyers, 123abc
```

Figure 3. Process Order View

responses are *continuation-passing* and promises [21]. In continuation-passing, a request carries a callback argument to which a response should be sent. Waterken and Causeway support both styles of response handling. Since continuation-passing is the dominant pattern for web application development, our example is written in this style. Waterken applications are written in Joe-E, a sequential subset of Java [22]. We limit our explanation of Waterken and Joe-E to what is required to explain the code snippets.

Our example program implements a procedure for handling new purchase orders. Before an order is placed, certain conditions must be met: the item is in stock and available, the customer's account is in good standing, and the delivery options are up to date.

An object residing in the "buyer" process has remote references to objects residing in the "product" and "accounts" processes. The buyer queries the remote objects with asynchronous message sends. Due to the limitations of Java syntax, in Waterken/Joe-E, the following statement sends the message and arguments asynchronously to `anObject`.

```
_._(anObject).aMessage(arg1, arg2);
```

For conciseness, in this paper we use the following notation.

```
anObject ← aMessage(arg1, arg2);
```

To collect the three answers, `teller` is passed as an argument to each of the remote inquiries, serving as a callback function.

```
Callback teller =
  new AsyncAnd(_, 3, new CheckAnswers());

inventory ← partInStock(partNo, teller);
creditBureau ← checkCredit(name, teller);
shipper ← canDeliver(profile, teller);
```

The answers from the asynchronous inquiries must be collected and examined. The order is placed only if all requirements are satisfied. The solution is an asynchronous adaptation of the conjunctive `and` operator, familiar from sequential programming, implemented below by `AsyncAnd`.

```
(1) public class AsyncAnd
(2)       implements Callback, Serializable {
(3)    private final Eventual _;
(4)    private int expected;
(5)    private Callback tellAreAllTrue;
(6)
(7)    public AsyncAnd(Eventual _,
(8)                    int expected,
(9)                    Callback tellAreAllTrue) {
(10)     this._ = _;
(11)     this.expected = expected;
(12)     this.tellAreAllTrue = tellAreAllTrue;
(13)   }
(14)
(15)   public void run(boolean answer) {
(16)     if (tellAreAllTrue != null) {
(17)       if (answer) {
(18)         expected -= 1;
(19)         if (expected <= 0) {
(20)           _.log.comment("happened: all true");
(21)           tellAreAllTrue ← run(true);
(22)           tellAreAllTrue = null;
(23)         } else {
(24)          _.log.comment("leadsto: all true");
(25)         }
(26)       } else {
(27)         _.log.comment("found a false");
(28)         tellAreAllTrue ← run(false);
(29)         tellAreAllTrue = null;
           }
         }
       }
     }
```

As each expected answer is collected, `AsyncAnd` counts down the total (18) and reports `true` only if `teller` sees `true` for every expected answer (21). A `false` answer short-circuits the logic. If `teller` sees `false`, `AsyncAnd` promptly reports `false` (28).

Besides send events, receive events, and call events, the programmer may indicate that something of interest has occurred by logging a *comment event* (27). Figure 2 shows the Lamport diagram for our example program. The turn boxes represent the sequential activity spawned by receiving a message, as explained in Section 3.

Causeway's process-order view (Figure 3) is a tabbed view, where each tab represents a process. For the selected process – in this case, the buyer process – it shows a 2-level tree of events in chronological order. The parent item represents a receive event; nested items are comment events and asynchronous message sends. In the next section, we argue that this process-order view is less helpful than one might expect.

2.2 Sequential Debugging Revisited

Consider the stack view in sequential debuggers. For most of us, the stack view is central to debugging sequential programs, but because of its familiarity, we don't see its excellent properties.

When the stack view shows the state of a sequential program, total causality is not shown. The call chain describes partial causality. Most of the time this is the interesting control flow describing what is going on. The call chain helps answer the question "What likely caused this to happen".

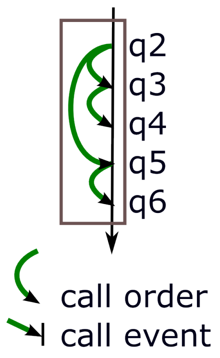


Figure 4. Call-return Order

Figure 4 shows a sequence of events in a single sequential process.

Let's say, to track down a bug, you set a breakpoint at q5. When hit, the sequential debugger shows a useful set of variables and the call chain.

In this case, at the q5 breakpoint, the call chain refers to q2. But the happened-before relation tells us that it's possible that q3 or q4 caused the bug. Wouldn't we necessarily find the bug faster if the debugger showed full process order?

Our experience with sequential debugging tells us, no – the partial causality shown by the call chain turns out to be the interesting causality in most cases. Usually, walking back the stack is enough. When this does not reveal the cause, you go backward in time to set a breakpoint. By going backward in time, you make use of the more detailed process order. In this case, you would set a breakpoint at q2 and run the sequence from the beginning, stepping through state changes.

To find the bug that manifests at q5, the first question to consider is what is the most significant influence on the current state at q5. The answer is most likely to be found in the chain of prior requests. By knowing to focus on q2 and to ignore q3 and q4, your debugging effort is put to good effect.

In the worst case, potential causality – the happened-before relation – must be considered. You must examine q2, q3, q4, and q5, although you can know that you can safely ignore q6.

Sequential debuggers, by presenting partial causality – the call-return order of a sequential program – support more effective debugging. Does this lesson generalize to distributed debugging?

2.3 Message-based Causality

Consider a bug that manifests itself at p4 in Figure 1. Event p3 happened before p4 simply because it occurred earlier in the same process. Event q5 happened before p4 because q5 expressed a request that p4 tried to satisfy. The lesson from sequential debugging is that we should start our search at q5.

Using Actor terminology, the call chain of a sequential program shows *activation order*. For distributed systems activation order includes local calls and distributed messages. By following activation order, you see the chain of requests expressing intended computational effects. Of course, buggy programs include unintended side-effects. But focusing on the intended causality first is an effective debugging strategy.

This observation, more than anything else, was the motivation behind Causeway: that activation order could be as significant to understanding distributed program behavior as it has proven to be for sequential programs.

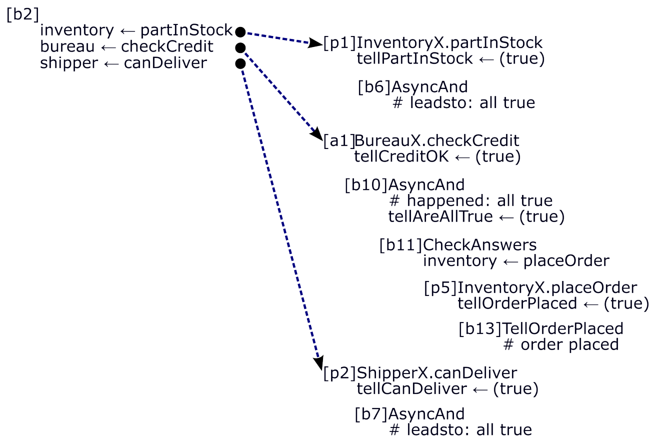
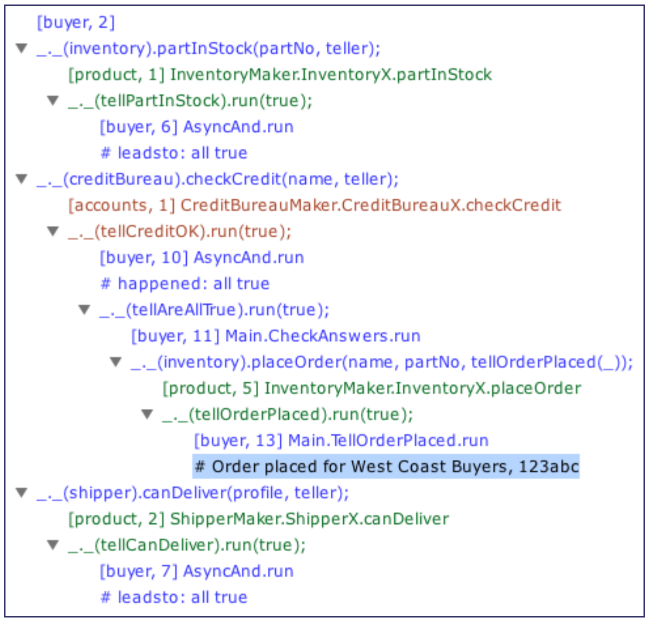


Figure 5. Message Order

Figure 5 shows the message events from the Lamport diagram of Figure 2 with the visual emphasis on message order. Causeway's message-order view (Figure 6) shows the order in which events caused other events by sending messages. The model is simple: a message is received and this causes other messages to be sent. The initial message is the cause; subsequent messages are effects. These effects can, in turn, cause later effects. Message order is reflected in the outline structure; nested events were caused by the parent event. Causeway assigns each process a color so we can see when message flow crosses process boundaries.

Here, we see that the bureau's reply to the buyer's query satisfies the final requirement. Meeting all conditions causes the order to be placed.



```
[buyer, 2]
▼ _._(inventory).partInStock(partNo, teller);
    [product, 1] InventoryMaker.InventoryX.partInStock
  ▼ _._(tellPartInStock).run(true);
      [buyer, 6] AsyncAnd.run
      # leadsto: all true
▼ _._(creditBureau).checkCredit(name, teller);
    [accounts, 1] CreditBureauMaker.CreditBureauX.checkCredit
  ▼ _._(tellCreditOK).run(true);
      [buyer, 10] AsyncAnd.run
      # happened: all true
    ▼ _._(tellAreAllTrue).run(true);
        [buyer, 11] Main.CheckAnswers.run
      ▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
          [product, 5] InventoryMaker.InventoryX.placeOrder
        ▼ _._(tellOrderPlaced).run(true);
            [buyer, 13] Main.TellOrderPlaced.run
            # Order placed for West Coast Buyers, 123abc
▼ _._(shipper).canDeliver(profile, teller);
    [product, 2] ShipperMaker.ShipperX.canDeliver
  ▼ _._(tellCanDeliver).run(true);
      [buyer, 7] AsyncAnd.run
      # leadsto: all true
```

Figure 6. Message Order View

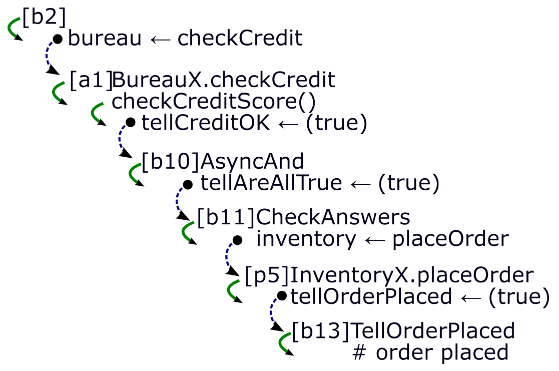


Figure 7. Activation Order

3. Communicating Event Loops

Causeway supports debugging distributed programs built on the communicating event loops concurrency model. In this model, a process consists of a heap of objects and a single thread of control. The control sequence combines the immediate call-return model of sequential computation with the asynchronous message-passing Actor model of distributed computation. Each object resides within one process. If objects *C* and *O* are in the same process, *C* can make requests of *O* using either immediate calls or asynchronous sends. If *C* and *O* reside in different processes, *C* can make requests of *O* using only asynchronous sends.

An asynchronous send of message *M* to object *O* residing in process *V* enqueues on *V*'s fifo queue a *pending delivery* recording the need to eventually deliver *M* to *O*.

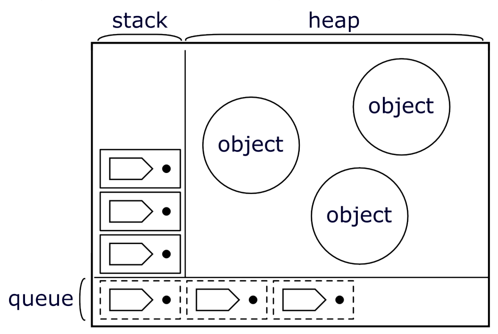
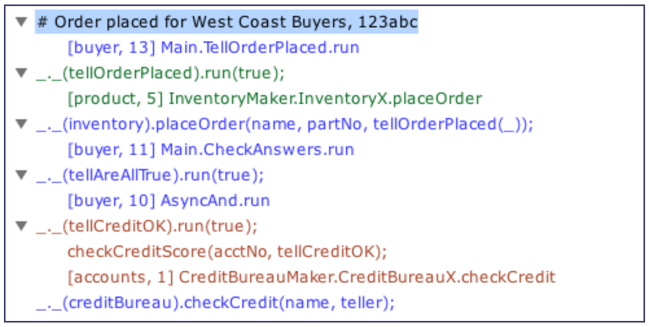


Figure 9. Event Loop Process State

Within its single thread of control, a process has both a normal call stack for immediate call-return and a fifo queue containing all the pending deliveries (Figure 9). Execution proceeds by taking a pending delivery from the queue, delivering its message to its object, and processing all the resulting immediate-calls in conventional call-return order. This is called a *turn*. A turn is an atomic unit of operation. When a pending delivery completes, the next one is de-queued, and so forth. The thread proceeds from top to bottom and then from left to right.



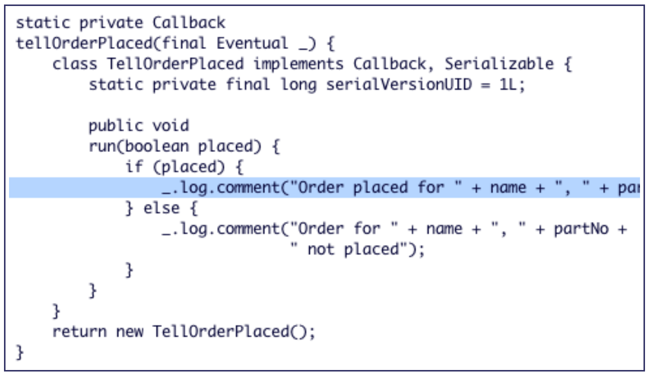
```
▼ # Order placed for West Coast Buyers, 123abc
      [buyer, 13] Main.TellOrderPlaced.run
▼ _._(tellOrderPlaced).run(true);
      [product, 5] InventoryMaker.InventoryX.placeOrder
▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
      [buyer, 11] Main.CheckAnswers.run
▼ _._(tellAreAllTrue).run(true);
      [buyer, 10] AsyncAnd.run
▼ _._(tellCreditOK).run(true);
      checkCreditScore(acctNo, tellCreditOK);
      [accounts, 1] CreditBureauMaker.CreditBureauX.checkCredit
  _._(creditBureau).checkCredit(name, teller);
```

Figure 8. Stack View

What determines a turn boundary? Each receive event starts a new turn and the turn continues with a sequence of local calls and send events until the call stack is empty. Message order forms a tree of turns. Within each turn, call order forms a tree rooted in its initial receiving event. As shown in Figure 7, the chain of requests leading to *[b13]* is the path backward through the combined activation tree.

Causeway's stack view (Figure 8) portrays the same information as Figure 7, but in the opposite order. As with sequential debugging, the question is often: How did we get here – what chain of activations led to the current event? The stack view helps answer this question by looking backward in activation order, presenting both asynchronous sends and immediate calls that led to the current event.

An entry in the stack view is a 2-level subtree. The parent item represents a send event; its nested items represent the stack trace captured for that event. The top entry is the currently selected event in the message-order view. Subsequent entries are built by following message order back in time to sending events. The two levels together show the path back through activation order. As an item is selected, the corresponding source position is highlighted (Figure 10).



```
static private Callback
tellOrderPlaced(final Eventual _) {
    class TellOrderPlaced implements Callback, Serializable {
        static private final long serialVersionUID = 1L;

        public void
        run(boolean placed) {
            if (placed) {
                _.log.comment("Order placed for " + name + ", " + pa
            } else {
                _.log.comment("Order for " + name + ", " + partNo +
                              " not placed");
            }
        }
    }
    return new TellOrderPlaced();
}
```

Figure 10. Source View

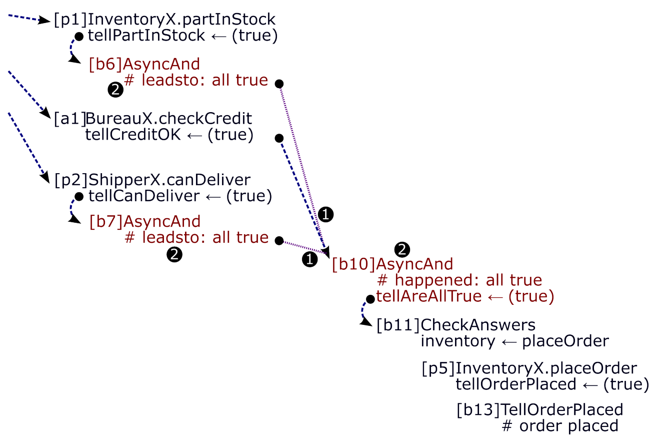


Figure 11. Promotion & Filtering

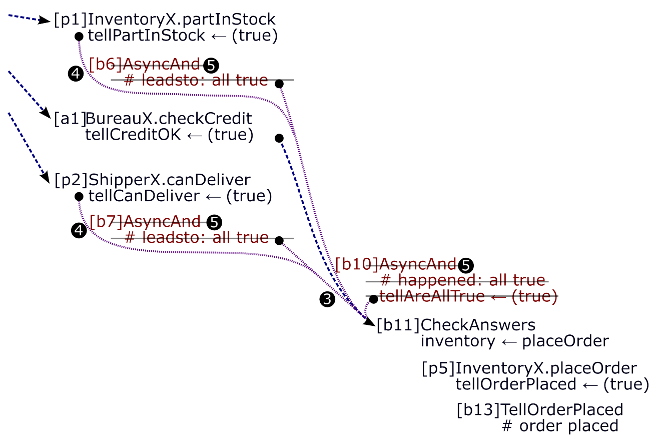


Figure 12. Aggregation

4. Event Abstraction

Causeway's process-order view (Figure 3) and message-order view (Figure 6) describe distributed behavior, giving an overview of program execution by showing how a set of events are related. But the overview is only as good as the set of events it describes. Non-trivial programs will obscure the overview with repeated patterns of irrelevant details.

As programmers, we use abstraction to hide detail. We might like our debugger to hide events internal to a reliable abstraction, presenting it instead as a higher level primitive. We can collapse pathways that go through event patterns by recognizing that a repeated event pattern is generally brought about by a piece of code representing a particular abstraction.

Of course, while debugging, we often want to see through some encapsulation barriers, to reveal imperfect abstractions. As our suspicions change, so should the abstraction boundaries respected by our debugger. The Causeway user controls which event patterns to abstract by choosing which bodies of code to treat as opaque black boxes, and which should be subject to open examination.

Causeway supports customizable event *promotion*, *filtering*, and *aggregation*. We describe each mechanism in isolation referring to Figures 11 and 12 (derived from Figure 5) to give a visual explanation of our algorithms. These algorithms are implemented by a post-order traversal, i.e., they proceed from right to left in our diagrams. Finally we show how these mechanisms work in concert to enable the programmer to tailor Causeway's presentation to match their current suspicions.

4.1 Event Promotion

It can be useful to highlight the significance of certain state-based causal relations by promoting them to *virtual* message order. `AsyncAnd` is a good candidate for event promotion. Let's say all expected answers are `true`. In this case, it reports `true` to its callback function and the order is placed. The message-order view (Figure 6) shows that the response to the credit check caused the order to be placed, but the programmer knows there were actually three causes. What happened to the other two contributors?

The first two answers change local state (a counter is decremented); the final answer causes a message send. Only the final answer appears in the message-order view. This correctly reflects how `AsyncAnd` works but not what it does.

Causeway's customizable event promotion builds on Waterken's support for logging comment events, as explained in Section 2.1. Notice the comments logged by `AsyncAnd`. The following comment is simply a string output – basically, a `printf` debugging statement.

```
    _.log.comment("found a false");
```

The following two comments demonstrate how the `AsyncAnd` author can indicate that one turn has significantly contributed toward the occurrence of a later turn.

```
    _.log.comment("leadsto: all true");
    _.log.comment("happened: all true");
```

The “`leadsto:`” prefix logs that the current turn has significantly contributed toward the possible occurrence of a later event of interest. The “`happened:`” prefix together with a matching suffix indicates that this event has occurred.

This comment format is not defined by Causeway; rather, it is a convention established between the application developer and the programmer customizing Causeway's event promotion. In step ❶ of Figure 11, we show the purple arcs added to promote the `leadsto:` events into secondary causes of the turn containing the matching `happened:`. Notice that if one of the inputs is `false`, no matching `happened:` will be logged, and the earlier `leadsto:` events will not be promoted.

In actual use, the comment records uniquely identify their `AsyncAnd` instance, so two overlapping but independent instances are promoted separately.

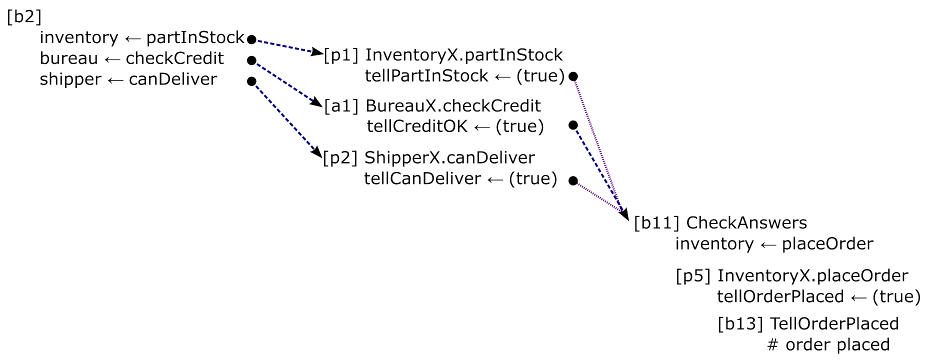


Figure 13. Abstracted Message Order

4.2 Event Filtering

Most debuggers support filtering infrastructure code and some allow selective filtering of user code [26] to hide stack frames executing in the filtered sources. Causeway's customizable filters likewise list the source paths that can be optionally shown or hidden.

Hiding stack frames supports customizable abstraction at the call-return level. Step ❷ of Figure 11 filters out the stack frames whose source is in the `AsyncAnd` class. When all the stack frames of an event are filtered out, that event becomes unlabeled, erasing the text shown in red on Figure 11. But even without event labels, the process and message-order views would continue to present distracting detail. Can we do better?

4.3 Event Aggregation

After filtering erases the red text in Figure 11, some turns will contain only unlabeled events, becoming empty. In Figure 11, all three turns containing red text happen to become empty, but this will not generally be the case.

When a turn is empty, it represents computation internal to the abstractions the programmer has already requested Causeway to hide. Assuming the filtered abstractions are working correctly, these turns do not contribute to understanding the code currently considered relevant. Our final abstraction step, *event aggregation*, erases empty turns when it can, while preserving the causal structure flowing through them.

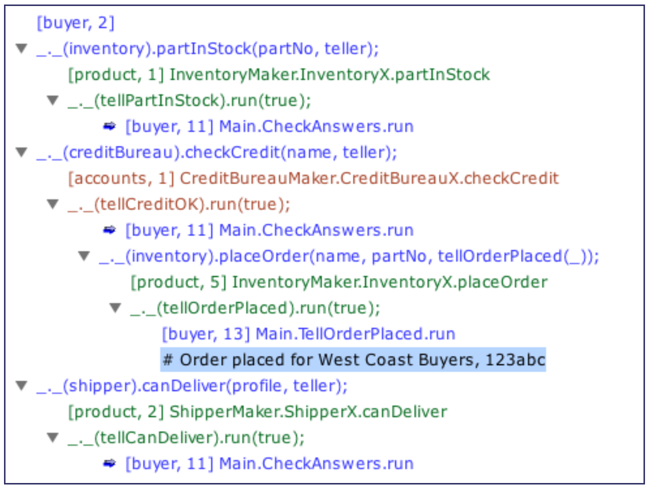
When an empty turn is a message-order leaf – when it causes no further turns in virtual message-order – then it can simply be erased. Any antecedent events are instead presented as causing nothing. When this results in other empty turns becoming leaves, they too are erased, etc.

After erasing all leaves, some empty turns are left causing only one further turn in virtual message order. These empty turns may also be erased with care. Again, in Figure 11, all three empty turns happen to cause only one further turn, but this will again not generally be the case.

For each empty turn causing only one further turn, we first splice all arcs into that turn to instead go into the only event that turn causes. In step ❸ of Figure 12, we first splice all three arcs into *[b10]* to instead go into *[b11]*. With the target of *[b6]* now recorded as *[b11]*, in step ❹, we splice the arc from *[p1]* to *[b6]* to instead go into *[b11]*; and likewise with the arc from *[p2]* to *[b7]*. Finally, in step ❺, we erase all uncaused empty turns. Turns *[b6]*, *[b7]*, *[b10]*, and *[b11]* are thus aggregated together into *[b11]*. The resulting graph is shown in Figure 13. The resulting message-order view presenting this graph is shown in Figure 14.

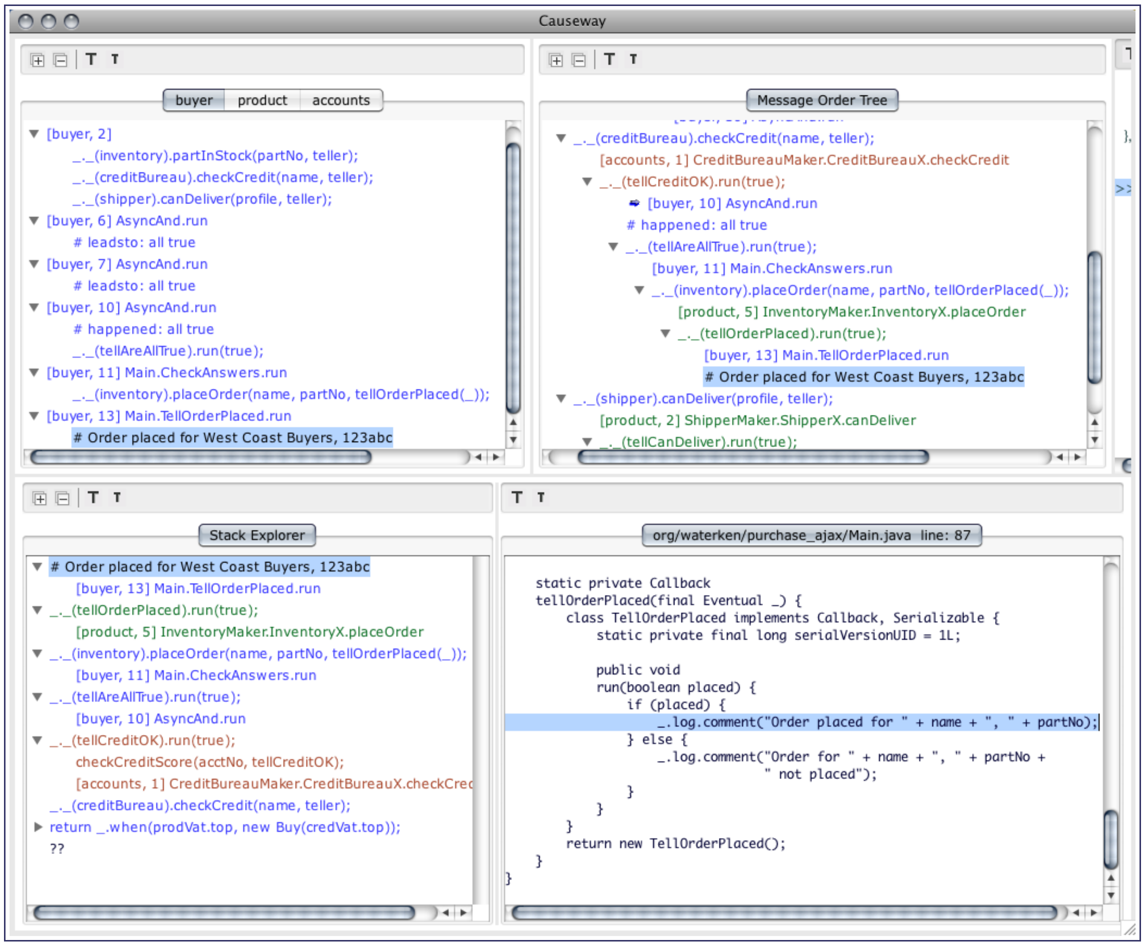
This algorithm will erase all inconsequential empty turns, but it will leave in place those empty turns that cause multiple other turns, as these carry important structural information. To erase them would require duplicating other arcs, which would be misleading.

The presented directed acyclic graph is still mostly tree-like, and the non-virtual message-based causes still form an actual tree. Notice that [buyer, 11] appears three times. Each occurrence is preceded by an icon which indicates that this event has multiple causes. Causeway's message order view presents virtual message order as an outline reflecting the primary tree, with these icons marking secondary causes.



```
  [buyer, 2]
▼ _._(inventory).partInStock(partNo, teller);
    [product, 1] InventoryMaker.InventoryX.partInStock
  ▼ _._(tellPartInStock).run(true);
      ➭ [buyer, 11] Main.CheckAnswers.run
▼ _._(creditBureau).checkCredit(name, teller);
    [accounts, 1] CreditBureauMaker.CreditBureauX.checkCredit
  ▼ _._(tellCreditOK).run(true);
      ➭ [buyer, 11] Main.CheckAnswers.run
    ▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
        [product, 5] InventoryMaker.InventoryX.placeOrder
      ▼ _._(tellOrderPlaced).run(true);
          [buyer, 13] Main.TellOrderPlaced.run
          # Order placed for West Coast Buyers, 123abc
▼ _._(shipper).canDeliver(profile, teller);
    [product, 2] ShipperMaker.ShipperX.canDeliver
  ▼ _._(tellCanDeliver).run(true);
      ➭ [buyer, 11] Main.CheckAnswers.run
```

Figure 14. Simpler Message Order View



```
▼ [buyer, 2]
      _._(inventory).partInStock(partNo, teller);
      _._(creditBureau).checkCredit(name, teller);
      _._(shipper).canDeliver(profile, teller);
▼ [buyer, 6] AsyncAnd.run
      # leadsto: all true
▼ [buyer, 7] AsyncAnd.run
      # leadsto: all true
▼ [buyer, 10] AsyncAnd.run
      # happened: all true
      _._(tellAreAllTrue).run(true);
▼ [buyer, 11] Main.CheckAnswers.run
      _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
▼ [buyer, 13] Main.TellOrderPlaced.run
      # Order placed for West Coast Buyers, 123abc
```

```
▼ _._(creditBureau).checkCredit(name, teller);
    [accounts, 1] CreditBureauMaker.CreditBureauX.checkCredit
  ▼ _._(tellCreditOK).run(true);
      [buyer, 10] AsyncAnd.run
      # happened: all true
    ▼ _._(tellAreAllTrue).run(true);
        [buyer, 11] Main.CheckAnswers.run
      ▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
          [product, 5] InventoryMaker.InventoryX.placeOrder
        ▼ _._(tellOrderPlaced).run(true);
            [buyer, 13] Main.TellOrderPlaced.run
            # Order placed for West Coast Buyers, 123abc
▼ _._(shipper).canDeliver(profile, teller);
    [product, 2] ShipperMaker.ShipperX.canDeliver
  ▼ _._(tellCanDeliver).run(true);
```

```
▼ # Order placed for West Coast Buyers, 123abc
      [buyer, 13] Main.TellOrderPlaced.run
▼ _._(tellOrderPlaced).run(true);
      [product, 5] InventoryMaker.InventoryX.placeOrder
▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
      [buyer, 11] Main.CheckAnswers.run
▼ _._(tellAreAllTrue).run(true);
      [buyer, 10] AsyncAnd.run
▼ _._(tellCreditOK).run(true);
      checkCreditScore(acctNo, tellCreditOK);
      [accounts, 1] CreditBureauMaker.CreditBureauX.checkCred
  _._(creditBureau).checkCredit(name, teller);
▶ return _.when(prodVat.top, new Buy(credVat.top));
  ??
```

```
    static private Callback
    tellOrderPlaced(final Eventual _) {
        class TellOrderPlaced implements Callback, Serializable {
            static private final long serialVersionUID = 1L;

            public void
            run(boolean placed) {
                if (placed) {
                    _.log.comment("Order placed for " + name + ", " + partNo);
                } else {
                    _.log.comment("Order for " + name + ", " + partNo +
                                  " not placed");
                }
            }
        }
        return new TellOrderPlaced();
    }
}
```

Figure 15. Causeway User Interface

5. Causeway Viewer

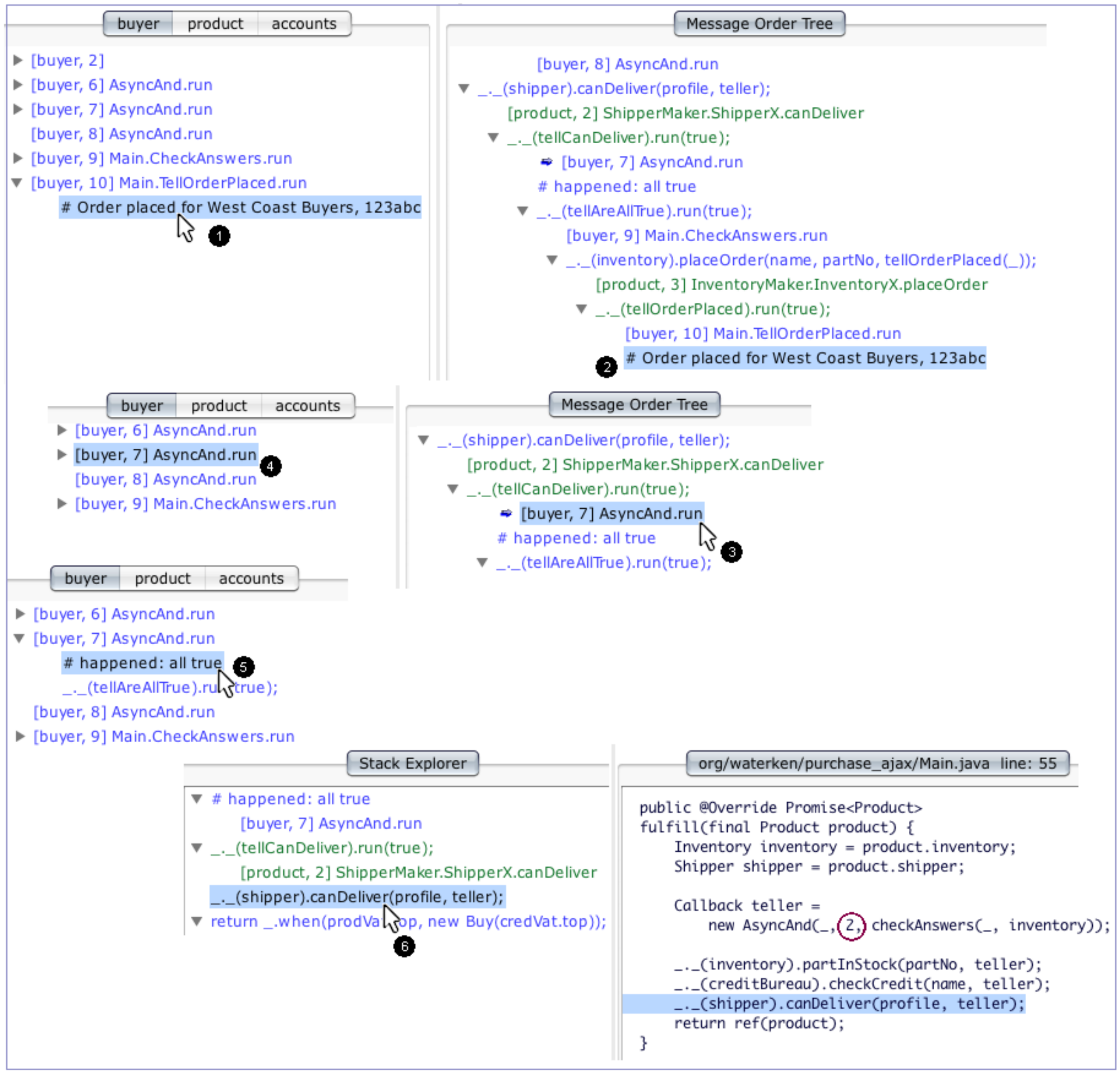
In this section we present Causeway's user interface. Figure 15 shows the principal views from Causeway's postmortem display. Causeway's views integrate multiple orders. The process and message-order views describe distributed behavior, giving an overview of program execution by showing how a set of events are related. The stack view and source code view are familiar from sequential debugging. They detail the internal structure of an event.

The views have been introduced in earlier sections. Here, we describe an interactive sequence for a debugging scenario. For concreteness, we introduce a bug into the simple purchase order program described in Section 2.1. Event promotion (Section 4.1) is enabled; event filtering (Section 4.2) is disabled.

The problem, as reported, is that an order was placed for a customer with bad credit. The search begins with the buyer process since it implements the controls for order processing. To see all events logged by the buyer process, select the buyer tab in the process-order view. This is the top, leftmost view in the interaction sequence described by Figure 16. Here we see a tree of events, in chronological order, for the buyer process.

The sequence of events shown seems to describe correct behavior: three responses to queries were received and the order was placed. But the order should *not* have been placed, so the next question is what led to the comment event, logging that the order had been placed. Select that event ❶ to get more information. Selection is synchronized across views, so the corresponding event is automatically selected in the message-order view ❷. Here we see the order in which events caused other events by sending messages. Message order is reflected in the outline structure; nested events were caused by the parent event. Message flow is followed across process boundaries. Causeway assigns each process a color so we can see when message order crosses process boundaries.

Scanning up the message tree, the `happened: all true` comment catches the eye. As explained in Section 4.1, this indicates that all conditions were met. To investigate this further, select its cause ❸. The corresponding event is selected in the process-order view ❹ and this looks odd – the order was placed after the second response rather than



```
▶ [buyer, 2]
▶ [buyer, 6] AsyncAnd.run
▶ [buyer, 7] AsyncAnd.run
  [buyer, 8] AsyncAnd.run
▶ [buyer, 9] Main.CheckAnswers.run
▼ [buyer, 10] Main.TellOrderPlaced.run
    # Order placed for West Coast Buyers, 123abc
```

```
      [buyer, 8] AsyncAnd.run
▼ _._(shipper).canDeliver(profile, teller);
      [product, 2] ShipperMaker.ShipperX.canDeliver
    ▼ _._(tellCanDeliver).run(true);
          [buyer, 7] AsyncAnd.run
          # happened: all true
        ▼ _._(tellAreAllTrue).run(true);
              [buyer, 9] Main.CheckAnswers.run
            ▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
                  [product, 3] InventoryMaker.InventoryX.placeOrder
                ▼ _._(tellOrderPlaced).run(true);
                      [buyer, 10] Main.TellOrderPlaced.run
                      # Order placed for West Coast Buyers, 123abc
```

```
▶ [buyer, 6] AsyncAnd.run
▶ [buyer, 7] AsyncAnd.run
  [buyer, 8] AsyncAnd.run
▶ [buyer, 9] Main.CheckAnswers.run
```

```
▼ _._(shipper).canDeliver(profile, teller);
      [product, 2] ShipperMaker.ShipperX.canDeliver
    ▼ _._(tellCanDeliver).run(true);
          [buyer, 7] AsyncAnd.run
          # happened: all true
        ▼ _._(tellAreAllTrue).run(true);
```

```
▶ [buyer, 6] AsyncAnd.run
▼ [buyer, 7] AsyncAnd.run
    # happened: all true
    _._(tellAreAllTrue).run(true);
  [buyer, 8] AsyncAnd.run
▶ [buyer, 9] Main.CheckAnswers.run
```

```
▼ # happened: all true
      [buyer, 7] AsyncAnd.run
▼ _._(tellCanDeliver).run(true);
      [product, 2] ShipperMaker.ShipperX.canDeliver
  _._(shipper).canDeliver(profile, teller);
▼ return _.when(prodVat.top, new Buy(credVat.top));
```

```
public @Override Promise<Product>
fulfill(final Product product) {
    Inventory inventory = product.inventory;
    Shipper shipper = product.shipper;

    Callback teller =
        new AsyncAnd(_, 2, checkAnswers(_, inventory));

    _._(inventory).partInStock(partNo, teller);
    _._(creditBureau).checkCredit(name, teller);
    _._(shipper).canDeliver(profile, teller);
    return ref(product);
}
```

Figure 16. Hunting for a Bug

the third. To dig deeper, expand the tree ❺. Now, to focus on a particular event, attention shifts to the stack view.

The top entry in the stack view is the currently selected event in the message-order view. Subsequent entries are built by following message order back in time to sending events. The two levels together show the path back through activation order. Expand the tree and select a message ❻ to see the corresponding source code in the source view.

We see that `AsyncAnd` is incorrectly instantiated to expect two answers, rather than three. The bug was not specific to the credit check, as initially thought. Whichever response happened to be received last, would be ignored.

Causeway is not a sequential debugger and the sequence described did not lead directly to the bug. Our purpose is to give a sense of the facility of moving between the process and message-order views to fathom the behavior of a distributed program. By showing how a set of events are related, these views give an overview of program execution. When focus shifts to a particular event, the stack view and source view give the event details.

Notice that synchronized selection between the process-order and message-order views is useful since, taken together, they convey the equivalent of Lamport diagrams (Figure 1).

6. Related Work

Debuggers can be categorized by their primary purpose and by the particular support they provide. Is a debugger used for understanding behavior for correctness or for performance analysis? Is it primarily for development debugging or production debugging? Distributed or sequential? Runtime or postmortem? Is it extensible? Does it support crossing various boundaries, such as user-kernel, component layers, or machine boundaries.

Causeway is a distributed, postmortem debugger used to understand program behavior for correctness, primarily during development and testing.

Postmortem debugging involves generating trace logs, collecting logged information, post-processing logs to extract relevant information, and the presentation of the resulting picture of the traced computation.

6.1 Generating Traces

Causeway's trace logs are so voluminous that their generation can affect system behavior. For debugging correctness issues during development, this cost is affordable. But debuggers aimed at production or performance analysis must do better.

DTrace [23] is an example of an engineering effort attempting to solve these problems. DTrace is a dynamic instrumentation framework integrated into the Solaris operating system. It is designed for performance analysis of both user and kernel-level software running on production systems. It supports user-controlled instrumentation that can be enabled and disabled dynamically. Users of DTrace can express *front-line filters* – filtering logic that runs at the point of trace generation – using DTrace's specialized programming language, *D*. Amoeba [25] similarly supports complex front-line filters to discard irrelevant trace records. Unlike DTrace, Amoeba supports distributed debugging.

Causeway's stack-frame filtering currently happens only during post-processing. When the programmer is confident enough in their lack of interest in some code, they could assign that portion of the filtering task to front-line filters supported directly by the logging system. Neither Causeway nor Waterken currently support this option. For stack frames suppressed by front-line filters, there would be no option to retroactively reveal them as the information is already lost.

6.2 Collecting Traces

Distributed systems face inescapable problems – partition, crash-revival of individual processes and servers, the participation of non-instrumented components, and the issues that arise when computing across administrative and trust boundaries. These problems can cause difficult bugs and they can also interfere with debugging. Each can prevent access to some log records. To tolerate partial logs, Causeway presents those causal relationships that it can derive from whatever logs are available. But this derivation is simplified by Causeway's assumption that all causality of interest is described in terms of the communicating event loops model.

X-Trace [24] is a dynamic, integrated tracing framework that gives a comprehensive view of the behavior of distributed components across layers and machines. The actual paths taken by data messages are traced across devices and layers. The construction of a complete task tree requires modification to clients, servers and network devices to propagate tracing metadata and to log reports. Like Causeway, X-Trace is tolerant of partial logs, though its reconstruction task is more challenging.

6.3 Extracting Relevance from Traces

Millipede [37] and IDLI [35] are built to support *multilevel debugging*, integrating the sequential view of conventional debuggers with support for *message level* and *protocol level* debugging. Causeway has no analog of their protocol level debugging, which attempts to determine whether logged messages conform to or violate a protocol specification. In Millipede and IDLI, log records are collected into a relational database. For those questions that can be formulated as database queries, this strategy can provide efficient answers even from voluminous logs.

Amoeba performs post-processing event abstraction using customizable *recognizers* – state machines matching meaningful patterns of events [25]. Only these abstract events are then presented by Amoeba's user interface.

Whereas Amoeba and Causeway only abstract events as directed by the user, Kunz enhanced the Poet debugger to "derive suitable abstraction hierarchies automatically." [20] Kunz constrains these automatically discovered abstractions to satisfy his *convexity* constraint, which is best explained by counter-example. If concrete event *X* happened before event *Y*, and *Y* happened before *Z*, then an abstract event *A* which aggregates *X* and *Z* but not *Y* would not be convex, since part of *A* happens before *Y* and *Y* happens before another part of *A*. By forbidding such non-convex abstract events, the resulting graph of abstract events can still be presented simply.

For communicating event loops systems, Kunz's convexity rule is safe but too conservative. If an inconsequential turn happened to occur in our example buyer process interleaved between two calls to `teller`, the convexity constraint would prevent the aggregation explained in Section 4.3. Instead, Causeway's aggregation algorithm only preserves Kunz convexity regarding message order taken by itself. This contrast demonstrates well why message order is generally more informative for reasoning about communicating event loops.

6.4 Presenting Traced Computation

Distributed debuggers invariably emphasize process order in their presentation of distributed computation. In [37] Kunz further enhances Poet to recognize logged *brackets*. As with Causeway's logged comments, these brackets were emitted by the program based on directives inserted by the programmer. Kunz provides an interactive visualization for forming abstract events by bracketing intervals along multiple parallel process order timelines.

While useful for Kunz's problem domain, this process order bracketing is inappropriate for communicating event loops. An independent turn that happens to interleave between two brackets must again prevent abstraction to avoid inclusion in the aggregate. The repeating event patterns needing abstraction are only stable when viewed in message order. Their interleaving with other events in process order is generally noise that should not inhibit abstraction.

Most distributed debuggers debug only the message level, seeing each sequential process as a black box [40] to debug using separate sequential debuggers. The dominant debugger for the browser side of web development, Firebug [39], does include both a conventional sequential debugger and some support for tracking AJAX messages. However, these two levels remain unintegrated. Firebug shows the request and reply headers for an AJAX message but does not capture the stack at the point of the AJAX call.

Sequential debuggers provide an integrated view of a program's sequential control flow together with access to its data state. Eclipse Step Filters [26] allow selective filtering of user and system-level code, and inspired Causeway's approach to source-based filtering.

Although Causeway does provide integrated views across both sequential and distributed control flow, Causeway provides no view of data state. To examine past data states, IDLI [35] logs enough information to deterministically replay individual processes, to provide the data access familiar from live debuggers, and do so for past states.

Waterken application are programmed in Joe-E, a deterministic subset of Java [36]. Were Causeway's logs enhanced to capture the additional information needed, Joe-E programs could be reliably replayed, perhaps using an enhanced Waterken server as a replay harness. However, a substantial engineering effort would be required to make this practical.

7. Discussion and Future Work

Communicating event loops present both benefits and new challenges for distributed debugging. Since each process is a sequential program sending only asynchronous messages, each program implicitly has mutually-exclusive access to all state to which it has synchronous access, so memory races are avoided without locking. Without locks or a blocking receive operation, conventional deadlock cannot occur. Most concurrency patterns can be implemented without concern for the bugs that plague multi-threaded programs. Debugging is simplified, as is the job of the debugger. However, less conventional race conditions and liveness bugs remain.

In Section 5 we saw a distributed consistency bug. Whether this bug manifests on a given run depends on a race condition – whether the bad credit report arrives before or after the second success. As we've seen, to track down a consistency bug, a good strategy is to start with an outcome that shouldn't have happened and look backward.

The `AsyncAnd` program presented in Section 2 contains an easily missed liveness bug: If the number of expected responses is zero, `AsyncAnd` should promptly report success. Instead, it fails to ever call its callback. When the discrepancy is that something never happens, there's no natural starting point from which to search backward. To track down a liveness bug, it usually works best to use the message view forward – to expand those branches along the path that should have led to the absent outcome, to see where it dead ends.

7.1 Methodology

Research on debuggers faces a methodological issue. Bugs purposely created in order to examine, such as our two `AsyncAnd` examples, are rarely realistic. Realistic undiagnosed bugs are typically encountered under pressures to get a program working rather than improve a debugger, so lessons learned are not recorded. Our experiences to date with Causeway bear this out.

- An early Causeway prototype built by Monica Anderson at Electric Communities helped us debug a massive distributed graphical virtual world [19]. Our memories of this experience are positive, but due to startup company pressures, we made no record of how this prototype did and did not help.
- To design the current Causeway user interface, we started with trace logs generated by running the E language's distributed command line loop. While hand simulating the visualization Causeway would generate from these logs, we easily found a bug that had resisted several earlier attempts at diagnosis [19].
- Next was the DonutLab challenge to build a complex distributed system over a weekend [33]. We did not have time to waste on bugs. With Causeway's help, we didn't. But again, due to time pressure, we made no detailed record.

As of this writing, a development project at HP Labs, ScoopFS [34], has a bug that manifests only when at least eleven server processes are involved. Though reproducible, this bug has eluded diagnosis for months. ScoopFS server processes are being upgraded to work on the version of Waterken/Joe-E instrumented to emit Causeway's trace logs. ScoopFS clients are currently programmed in ActionScript but will be converted to JavaScript. As we employ Causeway to search for this bug, we will record our experiences.

7.2 Future Work

If postmortem examination of ScoopFS server logs proves insufficient, several next steps are plausible. Once the

ScoopFS client is converted to JavaScript, instrumenting JavaScript in the browser to emit Causeway's logs will give us an integrated view of client-server interactions.

As mentioned in Section 6.4, we could jointly extend Causeway and Waterken to provide for instant replay of logged Joe-E computations, in order to provide access to past data states.

Our displeasure with process order is partially due to treating a process as a single undifferentiated mass of state – where any state change may affect any later turn in that process. With past data states available, we may be able to present a finer-grained view. For example, if two consecutive turns read and write only disjoint portions of their process' state, then it cannot matter which happened first in process order. Recognizing such independence would allow us to view process order as a partial order, enabling us to eliminate yet more events as irrelevant to a given bug.

Production systems present scaling problems. We do not know the practical scaling limits of Causeway, but suspect it is fairly small. We designed Causeway while thinking in terms of tens of processes, not hundreds or thousands. As use expands, we hope to understand this issue better.

8. Conclusions

The popularity of AJAX has introduced the difficult task of debugging distributed asynchronous programs to the mainstream programmer. Due to the nature of JavaScript in the browser, these programs execute as communicating event loops. This paper introduced Causeway, a postmortem debugger for distributed systems built on the communicating event loops concurrency model.

Causeway's novel message-oriented approach supports navigating the structure of distributed causality across process and machine boundaries. Its integrated views describe both distributed and sequential computation. We demonstrated how Causeway facilitates the search for consistency and liveness bugs, and discussed how users customize Causeway's event abstraction to hide detail currently considered irrelevant.

Causeway is open source and included in the latest version of E, available via subversion at http://erights.org/download/. Causeway recognizes a language neutral trace log format defined at http://waterken.sourceforge.net/debug/.

Acknowledgements

We thank E. Dean Tribble and Chris Hibbert for the initial ideas that led to Causeway, Monica Anderson for implementing an early Causeway prototype at Electric Communities, and Ka-Ping Yee for Figure 9 depicting the state of an event loop process.

We thank Jasvir Nagra, Mike Samuel, Toby Murray, Tom Van Cutsem, Alan Karp, and David-Sarah Hopwood for comments on earlier drafts of this paper.

References

[1] Johnson, L. W. Intention-Based Diagnosis of Novice Programming Errors. Research Notes in Artificial Intelligence. Morgan Kaufman. (1986).

[2] Waterken Server Documentation. http://waterken.sourceforge.net/

[3] Debugging a Waterken Application. http://waterken.sourceforge.net/debug/

[4] HTML5 postMessage. http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html#crossDocumentMessages

[5] Web Workers. http://www.whatwg.org/specs/web-workers/current-work/

[6] XMLHttpRequest Level 2. http://www.w3.org/TR/XMLHttpRequest2/

[7] XDomainRequest Object. http://msdn.microsoft.com/en-us/library/cc288060(VS.85).aspx

[8] Smith, David. Croquet Programming: A Concise Guide. Qwaq and Viewpoints Research Institute, 2006.

[9] John K. Ousterhout, 1994. Tcl and the Tk Toolkit. Addison-Wesley, Reading, MA, USA.

[10] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM, New York, NY, 230-243.

[11] David Mazières. A toolkit for user-level file systems. In Proceedings of the 2001 USENIX Technical Conference. pages 261-274, June, 2001.

[12] Kinder, K. 2005. Event-driven programming with Twisted and Python. Linux J. 2005, 131 (Mar. 2005), 6.

[13] Bracha, G. and Ahe, P. and Bykov, V. and Kashai, Y. and Miranda, E., The Newspeak Programming Platform.

[14] Mark S. Miller. "Robust Composition: Towards a Unied Approach to Access Control and Concurrency Control." PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[15] AsyncObjects Framework. http://asyncobjects.sourceforge.net/

[16] Dedecker J., Van Cutsem T., Mostinckx S., D'Hondt T., De Meuter W. Ambient-oriented Programming in AmbientTalk. In "Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP), Dave Thomas (Ed.), Lecture Notes in Computer Science Vol. 4067, pp. 230-254, Springer-Verlag.", 2006.

[17] Miller, M. S., Tribble, E. D. and Shapiro, J. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Trustworthy Global Computing, International Symposium (TGC '05)* (Edinburgh, UK, April 7-9, 2005). Springer Berlin / Heidelberg, 2005, 195-229.

[18] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM, Operating Systems, 21,* 7 (Jul. 1978), 558-565.

[19] Tribble, Distributed Space-Time Debugging http://www.eros-os.org/pipermail/e-lang/2002-November/007811.html

[20] Kunz, T. Visualizing Abstract Events. In Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '94), page 38, Oct. 1994.

[21] Hewitt, C., Baker, H. Laws for Communicating Parallel Processes. In 1977 IFIP Congress Proceedings, 1977.

[22] Mettler, A. M., Wagner, D. The Joe-E Language Specification. Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 17 2006.

[23] Cantrill, B. M., Shapiro, M. W. and Leventhal, A. H. Dynamic Instrumentation of Production Systems. Proceedings of the General Track: 2004 USENIX Annual Technical Conference (June 2004).

[24] Fonseca, R., Porter, G., Katz, R. H., Shenker, S. and Stoica, I. X-Trace: A Pervasive Network Tracing Framework.

[25] Elshoff, I. J. P. A Distributed Debugger for Amoeba. ACM SIGPLAN Notices, 24, 1 (Jan. 1989), 1-10. Special issue: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.

[26] Debugging with Eclipse Step Filters. http://www.eclipsezone.com/eclipse/forums/t83338.html

[27] Armstrong, J. Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, November 27 2003.

[28] Hoare, C. A. R. Communicating Sequential Processes. Communications of the ACM, 21(8):666–677, 1978.

[29] M Snir, SW Otto, DW Walker, J Dongarra, S Huss, MPI: The complete reference, 1995 - MIT Press Cambridge, MA, USA.

[30] Milner, R., Communicating and mobile systems: the pi calculus, Cambridge University Press, 1999.

[31] Kunz, T. and Seuren, M. F. H. Fast Detection of Communication Patterns in Distributed Executions. In Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '97), page 12, Nov. 1997.

[32] Garrett, J.J. and others, Ajax: A new approach to web applications, Adaptive path, 2005.

[33] Marc Stiegler, Mark S. Miller, Terry Stanley; "72 Hours to DonutLab: A PlanetLab with No Center"; Tech Report; Hewlett-Packard Laboratories; 2004.

[34] Karp, Alan H.; Stiegler, Marc; Close, Tyler, Not One Click for Security, HP Laboratories Tech Report HPL-2009-53.

[35] Basu, H. and Pedersen, J. B. IDLI: An Interactive Message Debugger for Parallel Programs Using LAM-MPI. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '06), Las Vegas, Nevada, USA, pages 513-520, June 2006.

[36] Finifter, M., Mettler, A., Sastry, N., and Wagner, D. 2008. Verifiable functional purity in java. In Proceedings of the 15th ACM Conference on Computer and Communications Security (Alexandria, Virginia, USA, October 27 - 31, 2008). CCS '08. ACM, New York, NY, 161-174.

[37] Tribou, E. H. and Pedersen, J. B. Millipede: A Multilevel Debugging Environment for Distributed Systems. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applicaitons (PDPTA'05) Las Vegas, Nevada, USA, pages 187-193, June 2005.

[38] Hauswirth, M., Diwan, A., Sweeney, P. F. and Mozer, M. C. Automating Vertical Profiling. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA), pages 281–296, Oct. 2005.

[39] Lerner, R. 2007. At the Forge: Firebug. Linux J. 2007, 157 (May. 2007), 8.

[40] Cheung, W. H., Black, J. P. and Manning, E. A Framework for Distributed Debugging. IEEE Software, 7, 1 (Jan. 1990), 106-115.