Implementation of causal control operators to detect distributed events

Luis Daniel Benavides Navarro
Escuela Colombiana de Ingeniería
Julio Garavito
luis.benavides@escuelaing.edu.co

Oscar Kiyoshige Garces
Universidad de los Andes
Bogota, Colombia.
ok.garces10@uniandes.edu.co

Hugo Arboleda and David Durán
Universidad Icesi. Cali, Colombia
hfarboleda@icesi.edu.co,
duran.david19@gmail.com

***Abstract*—This paper discusses the design and implementation of three operators to control and predicate about causality relation between events in a distributed system. Concretely, we start from KETAL, a kernel conceived to detect patterns of distributed events, and we motivate the need to keep track of causality relations between events on m-health applications over Wireless Body Area Networks (WBAN). We then introduce a causal event model and we present a detailed implementation that allows the notion of causality in KETAL. This implementation is based on vectorial clocks, and it supports the detection of concurrent and causal relationships, plus the dynamic administration of the nodes involved in the system.**

Keywords: Event-driven programming, e-health, m-health, Wireless Body Area Network, WBAN, Complex Event Detection, Automata, Distributed Systems, Causality, event ordering.

I. Introduction

Distributed systems use various communication protocols to perform message exchange between different devices. Those protocols grant data processing, computation, and result gathering in distributed algorithms. Exchanged messages contain information of distributed events occurred on each device, e.g., a user input, a file update, a battery depletion, a new device being connected to the network. A significant challenge concerning the evaluation and manipulation of these distributed messages is to guarantee a coherent exchange between them within a system, namely, to ensure that message reception occurs in the same order that they are generated.

There have been several proposals in previous works, see [1], [2], [3], [4], [5], [6], about the explicit manipulation of events in distributed applications by using event models, in particular, to control the flow of events generated inside distributed systems. In this paper, we investigate the design and implementation mechanisms for the explicit manipulation of message ordering and events. Concretely, we start from the KETAL [6] library, which presents an event-based solution that provides mechanisms to manipulate and detect event sequences through automata, and we extend it with mechanisms to allow the orderly manipulation of distributed events, conceiving the following contributions: An event model for causal manipulation of events and sequences of events, the design of three operators for the causal manipulation of ordering in event sequences, and a detailed description of the implementation of these three new operators.

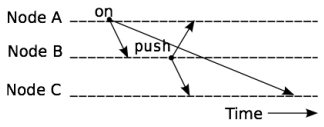

Fig. 1. Non-determinism affecting messages

This document is structured as follows: In section 2 we emphasize on the relevance of event ordering in distributed systems and we introduce two common problems concerning message exchange in those systems; in section 3 we describe the event model proposed in KETAL; in section 4 we characterize the causality model that is going to be incorporated to KETAL library; in section 5 we exhibit strategies used to implement the causality protocol; in section 6 we present an evaluation using a case study and, in section 7, we expose the state of art and related work to finally conclude and suggest future work.

II. Motivation: Messages and event ordering

Consider for example Fig. 1, the figure depicts a distributed application that has three nodes (computers). Each computer has a deployed application that simulates a stack behavior. This application keeps a copy of the stack in every node, replicating by means of distributed messages any modification to the remaining nodes. The simulated stack has the following protocol: before inserting or extracting any element (push/pop respectively), this structure must be turned on with a message.

In Fig. 1, Node A generates and processes an `on` event, which is replicated to the remaining nodes. Here we can see in detail the problem of non-determinism caused by the order of events arrival. In particular, consider that in a distributed application, the network is a fundamental component and it doesn't guarantee the order of the messages. Once Node B receives the on message, it adds a new element to the stack through a push event. Meanwhile, events generated by Node A and Node B are altered by factors that are external to the distributed replication algorithm but inherent to the network, i.e. communication delays, that cause reversal in their arriving order to Node C. This situation generates the invalid sequence of events `push-on`.

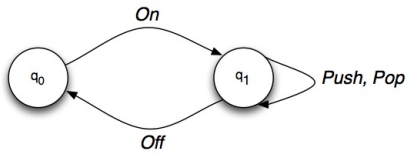

Fig. 2. Automata to model an stack

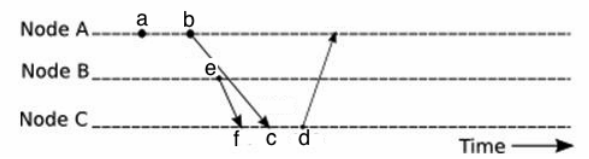

Fig. 3. Happens-before relationship

Sequence detection problems, as the one described above, grow exponentially in concurrent and distributed applications of industrial size. We classify these problems in two groups, false positives and false negatives. False Positives are event detection errors where the distributed system (the term distributed system refers to a set computers and applications running as a single system) recognizes a sequence of events that actually never occurred. False negatives are errors that do not recognize event sequences that actually did occur. Such issues have been previously documented in literature, showing how they affect the development and maintenance on big scale applications (see [2]).

III. Event and pattern models in KETAL

In this section, we briefly introduce the basic concepts of the KETAL library, which allows detection of event patterns by using finite deterministic automata. Next, we describe the adopted event model, the automata model to detect event sequences, and the guard model presented by KETAL.

A. Event Model

Here we start from an event model that has two essential elements: events and messages. An event is an atomic action that occurs within a process, application or system, i.e., execution of a function or method, action of sending and receiving messages, mouse clicks captured by a computer, etc. In a distributed context, the information of the occurrence of these events is communicated between devices through messages.

B. Automata to detect event sequences

To detect a specific event sequence, KETAL uses finite deterministic automata. In particular, each automaton will consume events as elements that belong to a recognizable language, allowing modeling and detection of event patterns. If we recall the stack situation, it is possible to model the desired behavior using four actions: On, Push, Pop, Off, for instance, allowance of Push and Pop events after the occurrence of an On event, as shown in Fig. 2. Please notice that this automaton still allow some invalid sequences, such as the occurrence of a Pop event without a previous Push event, situation that will generate an exception. This issue will be addressed in later sections, taking into account causal relationships between generated events.

C. Guard model

Besides the automata model, KETAL offers a guard model that enable developers to predicate over events. In particular, it allows predicating on the context of an event. For instance, using KETAL we can create an automaton that not only detects an On event, but also recognizes if such event occurred in a specific node. The expression would be like: Call(* ReplicatedStack.on(..)) && host("A"). In this article, we won't go into more detail about language syntax, because we only intend to highlight the existence of a guard model that will use the operators for causality management that we will build in later sections.

IV. Causal event model and causality operators

In this section, we present an extended model of KETAL, in which we incorporate the causality relationship defined by Lamport [7] and Mattern [8]. This relationship allows definition of predicates to detect complex and sophisticated event sequences in distributed applications.

A. Adding causal relationship to KETAL's event model

Event model presented in previous sections doesn't contemplate an order relationship between events occurred in distributed systems. Thus, those events were consumed in their arriving order by the automaton. In order to extend KETAL with a notion of order between events, we start from previous works about logical time, virtual time and the global state of distributed systems, particularly, work from [7], [8], where they design mechanisms and algorithms to establish a partial order in distributed event sequences. We then introduce the causality relationship that determines when an event has causal influence over another. To understand this connection, let's take a look at the Happens-before relationship defined in [7], which states that a causal relationship must meet any of the following cases. Let `a`, `b` and `c` be events:

- `a` and `b` are in the same process and `a` takes place before `b`; then `a` happens-before `b` (`a` is causally related to `b`).
- `a` represents an event of sending a message while `b` is an event of receiving a message, then `a` happens-before `b`.
- If `a` happens-before `b` and `b` happens-before `c`, hence, `a` happens-before `c` due to relationship's transitivity.

In any other case, events are considered concurrent ([8]). To illustrate the happens-before relationship, let's analyze the situation described in Fig. 3. Because `a` and `b` are in the same process or node, and `a` occurs before the `b`, we declare that `a` happens-before `b`. On the other hand, `c` is a receiving event on Node C, of a message sent by Node A, hence, `b` happens-before `c`. The same situation applies between `e` and `f` events. Following the third condition, we can deduce that `a` happens-before `c`. In this scenario we cannot establish a relationship between `b` and `e` in terms of causality, therefore these events

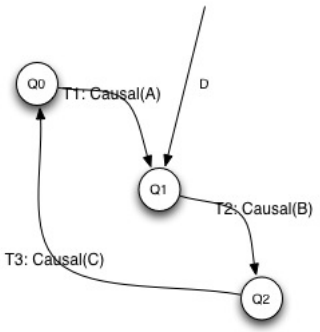

Fig. 4. Causal automata interaction

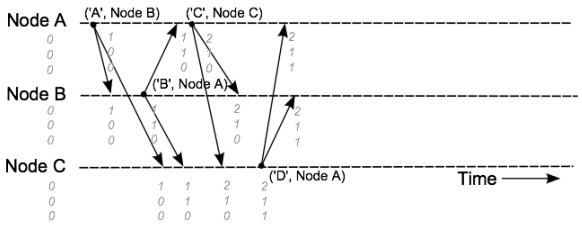

Fig. 5. Messages exchange between nodes

are concurrent. Thus, the event model states that two events are causally related if a happens-before relationship can be found between them.

B. Operators to support causal relationships

In this work, we propose three operators to manipulate causality between event sequences. The two first operators apply to events and will be denominated Causal and Conc. Former indicates that a transition in an automaton will occur if and only if the detected event is causally related to the immediate prior event. Syntax for this operator is Causal(event), where event represents an event we are interested in. Later operator has Conc(event) syntax, which states that an event will be consumed if and only if it has a concurrent relationship with the immediate prior event. The third operator applies to the application and doesn't establish a concrete syntax; however, in this article we will denominate it CausalOrder. When this operator is inactive, events within the system won't follow a particular order, thus they will be consumed in their arrival order. Conversely, if CausalOrder operator is active, system will take into account causal order and the automaton will process events according to this criterion. Fig. 4 depicts an example of the behavior of sequence detection under operators influence. The automaton shown starts in Q0 state and has to detect an event A that has to be causally related to the previous event consumed. If A is the first event being consumed by the automaton, causal order will be ignored. Later, the automaton is programed to consume an event B that must be causally related to the previous event consumed and, finally, in order to consume event C, it has to satisfy a causal condition with its predecessor as well. The figure also shows an alternate way to get to Q1 state through event D, which implies that event B has to be causally related to D in order to fulfill the next transition's constraint. In Fig. 5 we can concretely appreciate the interaction and dynamics of transitions between states.

Let's assume that we are in the initial phase of a system and each node has a copy of an automaton programmed to start in Q0. Fig. 5 shows that event A occurs in Node A, which triggers a transition to move from Q0 to Q1. Later, event D is detected and does not trigger any transition, but it is stored in a temporary memory. When event B occurs, transition from Q1 to Q2 is triggered, because B and A are causally related. The next generated event is C, which triggers a transition to get back to the initial state Q0. Finally, event A does take place in node A, as shown in the figure. Event A is not causally related to event C, so when the automaton evaluates A, it won't change its state because these events are concurrent. This is how an automaton will process event sequences that happen within a distributed system.

V. Implementation

This section discusses causality implementation details in KETAL [6]. This event-driven library focuses on modeling and detecting event patterns occurred within a system, enabling posterior manipulation of detected events, allowing a deeper analysis of non-deterministic complex systems. As mentioned before, there is a distinction between events and messages inside distributed systems. An event is an atomic action occurred in a specific process, e.g., a Java method invocation, sending and receiving a message. A message is responsible for encapsulating and transporting event information through a distributed system.

A. KETAL's architecture

Fig. 6 depicts KETAL's architecture, which has four main components: Abstract Event Framework, Automata Facade, Automata Engine and JGroups Extension Layer. Abstract Event Framework contains a set of interfaces that allow software developers to model an entire scenario of interest using events, plus the use of automata to detect event patterns. The whole functionality to set up a state machine and to process a defined automaton lies upon the Automata Facade component. Automata Engine is based on Anders Møller [9] library, which contains an implementation of a deterministic finite automata engine. Last component provides core abstractions to distribute messages among nodes. The implementation of this component is an extension of JGroups library [10].

KETAL's event model is represented in Fig. 7, which includes the following features: i) The model is not coupled with the libraries used for automata processing and distribution, i.e., it has a well-defined interface, ii) it is scalable and capable of detecting complex events through event expressions that are used in the automaton transitions. Current model provides basic And/Or binary Boolean operators. It also includes the Unary interface in order to support unary boolean operators

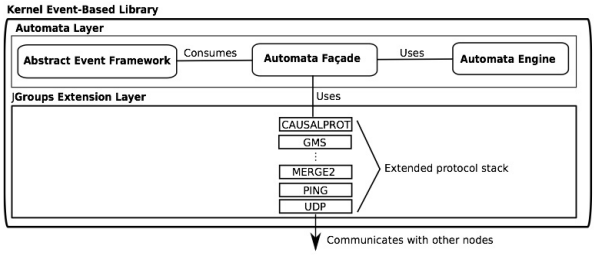

Fig. 6. KETAL's architecture

like Not. A transition between two states of the automaton will be executed if the Boolean expression asserts true. The Unary interface allows building expressions with unary operators. An expression of this type contains only one event, but it could take into account additional information, like the location of the host where a specific event occurred (i.e. ExecutionHost) or denial of an event (i.e. NotExpression). iii) Its flexibility allows developers to include any new Boolean operator needed to manipulate event expressions.

B. Causality Support

Our approach supports causality by adapting the algorithm designed and implemented by Mattern [8], which uses vectorial clocks to operate. Proposed algorithm defines a partial order of messages based on causality, and provides guard support to identify event relationships. The following example explains its operation. Fig. 8 depicts the interaction of three nodes. Each black point represents an event and each arrow represents a message. Each node contains a vectorial clock where each position references a node, thus positions 1, 2 and 3 correspond to Node A, Node B and Node C, respectively. Each occurrence of an event implies an update of the value contained in a position of the clock, depending on the node where it took place. Take the two first events on Node A for example, every event adds one to position 1, so, when they both occur, the vector will have values <2,0,0>. Situation on Node C is different. The first event is a reception of a message, so the vector will be <0,0,1> at first. The following step in Node C consists of comparing the current vector with the one in the arriving message, so if there is any position that contains a higher value than the current one, the higher value is copied and the current vector is updated. In this case, the arriving vector contains 1 in position 2, so vector on Node C is updated to <0,1,1>. Same situation happens with second event on Node C, where before comparing, the vector will be <0,1,2>, but after comparing with the arriving vector, it will be updated to <2,1,2>.

In order to enable detection of causal relationships using Mattern's algorithm, we follow the architecture illustrated in Fig. 9. This diagram gathers all components used by JGroups to enable their causality protocol. Here we can see that each Message carries in its CausalHeader a vectorial clock denominated TransportedvectorTime. JGroups defines a stack of protocols that split application functionality into layers. Each layer adds its own control information to messages exchanged to ensure they are treated properly, i.e. Causal protocol ads a vectorial clock (TransportedVectorTime) to each message header. Layer division uses labels to separate processing responsibilities into levels, so each level will only process corresponding data. This allows developers to add as many layers as needed to ensure desired behavior. We made several modifications to base JGroups implementation in order to extend its functionality and to adapt it to KETAL's event model. First alteration focused on allowing the association between KETAL and vectorial clocks to let the event model decide about occurrences. Second important change was to provide services to enable the use of causality operators described in section 4.2. Resulting implementation is described below.

Fig. 10 illustrates how vectorial clocks determine causality between events. Initially, every node has a vector equal to <0,0,0>. Later, event a occurs when Node A sends msg 1 to the remaining nodes, so vector in Node A is set to <1,0,0>. This updated vector is included in msg 1 header. Event b refers to reception of msg 1 on Node B, so vector in this node will be <1,1,0>, because it is updated with the arriving vector considering occurrence of b. Same situation happens on Node C, resulting in a vector equal to <1,0,1>. To determine if two events are causally related, a comparison between arriving vector and local vector must be made. If all values contained in local vector are greater than or equal to all values in arriving vector, then events are causally related. According to the previous condition, a and b are causally related, because all values in Node B's vector are greater than or equal to values in msg 1 vector; this is an example of the first situation described in section 4.1. The subsequent situation happens between events e and g, where g takes place after e, and g's vector is greater than or equal to e's vector. Transitivity of causal relationship can be illustrated between events a and k, because a happens before c, c happens before f, f happens before h, and h happens before k. Notice that it is possible to causally relate a and k with the alternate path a-b-d-e-g-h-k, depending on event occurrence. Events b and c depict a concurrent relationship because none of the situations explained in section 4.1 apply to them, plus their vectors don't meet the requirement to be causally related.

C. Dynamic management of nodes

Besides previous modifications, we had to consider certain functionalities related to dynamic behavior in a distributed environment, e.g., adding or removing nodes to the system implies updating vectorial clocks of each remaining node. The protocol supports this behavior by means of a mechanism that updates the number of existing nodes that are in the TransportedVectorTime vector clock on every host; however, to perform such operation certain considerations have to be taken into account, especially in a multithreaded system that cannot be paused. To address this issue in multithreaded environments, where availability is a crucial factor, the system instantiates an independent thread identified as NewViewThread, whose

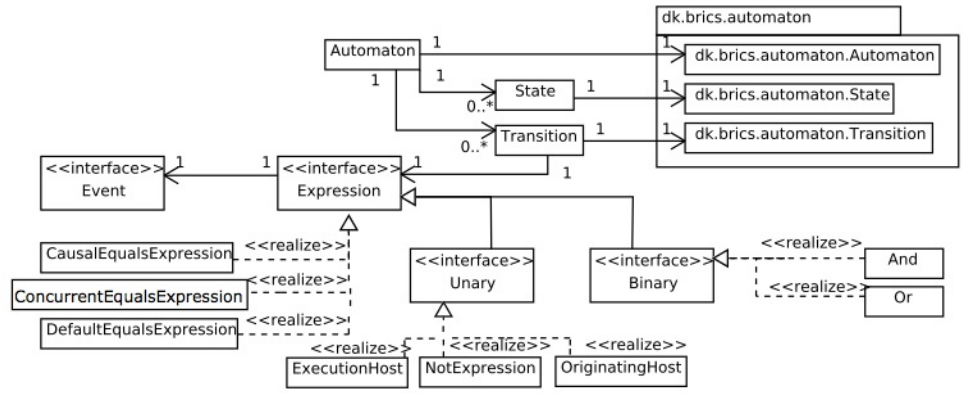

Fig. 7. KETAL's class model

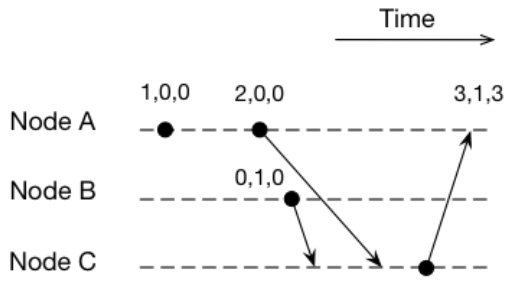

Fig. 8. Vectorial clock in a distributed system

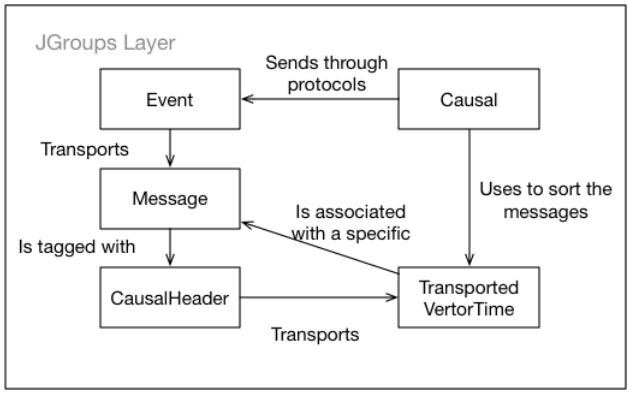

Fig. 9. Causal protocol architecture

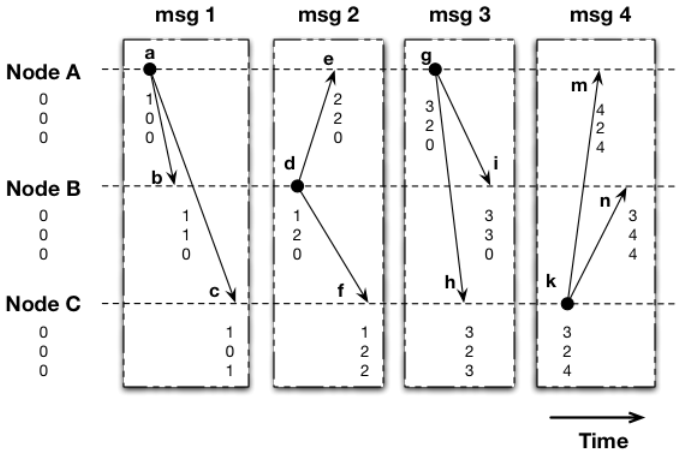

Fig. 10. Vectorial clock updates

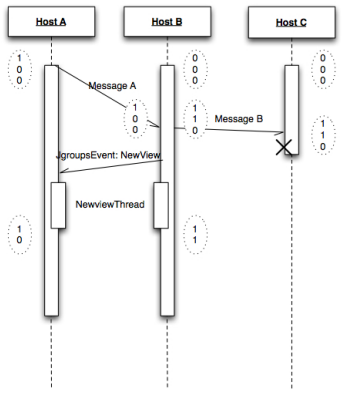

Fig. 11. Vectorial clock update with causal protocol sequence diagram

task is to keep an updated copy of the needed context information (e.g. current vectorial clock with updated nodes count and pending messages to be processed), while main thread continues receiving and manipulating events. Synchronization of these two threads ensures updated information to avoid protocol breakdowns. Fig. 11 contains a sequence diagram of a three-node scenario that uses KETAL and JGroups to handle message exchange. In this particular case, we are interested in how vectorial clock is modified once a node is retired from the system. Thus, while there is a main thread managing incoming and outgoing messages, there is one thread of type NewViewThread that takes a copy of current vectorial clock and eliminates the index in the position referencing the retired node. Once the vector is updated, the vectorial clock contains as many positions as nodes remain in the system.

D. Operators to guarantee detection of event with deterministic causal relationships and ordered detection of event

To implement causal relationships, we extended the expression hierarchy (see Fig. 7). Two expressions were conceived to implement operators: CausalEqualsExpression and ConcurrentEqualsExpression, to detect causal and concurrent relationships. Both of them compare an arriving event of type Event with the event used in the definition of the guard, but the former asserts true only if arriving event is causally related to the immediate prior event evaluated in the automaton. Later operator asserts true if arriving event isn't causally related to immediate prior event evaluated in the automaton, i.e., it is concurrent to the immediate prior event. The main difficulty during implementation laid on component interaction, especially message distribution. To address this issue we decided to use JGroups causality implementation, which establishes an event causal ordering, but it doesn't ensures causal relationships detection; so we had to adapt JGroups library in order to support vectorial clocks and causality management, without the need to order events exchanged. These modifications allow developers to use the event model and operators to detect causal relationships.

VI. Evaluation

To evaluate the validity of our proposal, we analyze behavior of a replicated and distributed stack. Assume that the stack is deployed on two nodes (two computers). The protocol of the system is quite simple; the stack must be turned on locally in order to process push and pop messages, each push and pop message is replicated to every participating stack in the distributed system. A participating stack deployed on a particular node may be turned off locally. In that case, such node is no longer involved in the distributed messaging. Consider now that a developer wants to debug the correct behavior of the replicated stack protocol. Thus, the proposed library may be utilized to implement an automaton that monitors such behavior. The developer has to define an automaton and deploy a copy of such automaton on each node. Fig. 12 contains the definition of a deterministic finite automaton that models the correct (desired) behavior of the system: in the first step the automaton expects a local On event, once automaton has changed to q1 state, push and pop events are accepted by the automaton, only if they are causally related with the On event. In the next transition, from q1 to q0, we have the local off event, which has to be causally related with any other previous event, especially with the on event. Causal relations in the automaton definitions grant that false positives are avoided. False negatives will be avoided only if the system is ordering messages, using the causal partial order, before consumption by the automaton.

Figure 13 shows the implementation of the automaton. First, we define two states using the State object of the library. We then define the three transitions giving them a name; in this case the names used are of type Char and have the values A, B, C. Those names are the alphabet of the automaton, and are used to reference complex events; they are especially useful

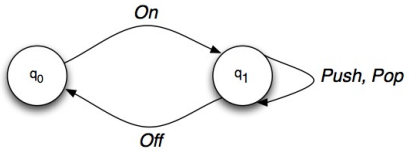

Fig. 12. Automaton to model a replicated stack protocol

to define automaton behavior using regular expressions. The transitions are grouped in a HashSet. Then we map each transition with a complex expression over events. Concretely, we first define an expression that accepts an event and compares it with the source event On (class DefaultEqualsExpression), the expression also checks whether the event is local or not. We then define an Or expression that accepts and event and compares it with a push or pop event, with the restriction of having a causal relation with previous events. In order to make such comparison we use CausalEqualsExpression instead of a DefaultEqualsExpression. Finally we define and automata using the set of states, the transitions set, the initial state, and the mapping of transitions to transition names. At runtime the automaton detects the correct behavior of the protocol. If any event arrives in the wrong order the automaton halts. Thus, the automaton detects a violation of the protocol. The library is designed to be a part of frameworks and languages that are based on events and provide syntactic tools (e.g., IDE) to define concisely automata and event expressions.

VII. State of the art

There are different approaches in literature regarding event paradigm in distributed systems. There are some focused on event ordering, others deal with creation of distributed middleware, and there is also a community considering event pattern detection using event-oriented programming as the proper way to deal with these topics. Solutions like [7], [8] present theoretical models to cope with causal ordering in distributed systems. On the other hand, [10] offers an implementation that enables causal ordered message distribution among nodes of a distributed system, generating a convergence between message distribution and message ordering. Proposals like [9] admit pattern recognition of symbols from a specific language, which was adapted and improved in [6] as an event detection solution in distributed systems. Later, solutions like [2] concentrates on event detection solution in distributed systems, with certain limitations on operator specification between events and concrete event-oriented language syntax. Finally, we find that relevance of causality operators approaches for event detection in distributed systems makes possible the convergence of message distribution and message ordering, plus it proposes a method to declare expressions with an event-oriented language to capture specific event patterns.

VIII. Conclusions

This article presented design and implementation of a causality model with operators conceived to detect causal and

```
1  /* Automaton states are initialilized */
2  State q0 = new State();
3  State q1 = new State();
4
5  /* A -> On, B -> Causal(Push) or Causal(Pop) */
6  Transition t1 = new Transition(q0, q1,    A   );
7  Transition t2 = new Transition(q1, q1,    B   );
8  Transition t3 = new Transition(q1, q0,    C   );
9  Set<Transition> transitions = new HashSet<
       Transition>();
10 transitions.add(t1);
11 transitions.add(t2);
12 transitions.add(t3);
13
14 /* Expressions to char mapping; chars are used
       as logical names of transitions*/
15 HashTable<Expression, Character> expressions =
       new Hashtable<Expression, Character>();
16 expressions.put(new And(new
       DefaultEqualsExpression(new TransactionEvent
       (   On   )), new OriginatingHost(
            localhost    )),    A    );
17 expressions.put(new Or(new
       CausalEqualsExpression(new TransactionEvent(
          Push    )), new CausalEqualsExpression(new
        TransationEvent(   Pop   )),    B    );
18 expressions.put(new And(new
       DefaultEqualsExpression(new TransactionEvent
       (   Off   )), new OriginatingHost(
            localhost    )),    C    );
19
20 /* Automaton initialization with transition set,
        an initial state, an expression set, null
       for set of final states */
21 Automaton transactionAutomaton = new Automaton(
       transitions, q0, null, expressions);
```

Fig. 13. Replicated stack automaton setup

concurrent relationships between distributed event sequences. The model uses a library to detect complex event sequences by using finite deterministic automata. We also expose our motivation, regarding incorrect event sequences detection caused by false positives and false negatives, to determine our causality model based on happens-before relationship defined in [7]. Once our model was defined, we presented an implementation based on vectorial clocks to support causality in KETAL. This implementation provides causal and concurrent operators, besides dynamic management of nodes. Finally, in evaluation section we exhibit flexibility and simplicity of automata setup and we illustrate how to manage causality, granting the developer freedom to model any kind of event sequence in complex systems. This proposal enables future development of error detection apps, e.g., debuggers, along with systems designed to run in distributed environments, just like the cases in section 6.

REFERENCES

[1] Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud. Invasive patterns for distributed programs. In *Proc. of the 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA'07)*. Springer Verlag, November 2007.

[2] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 183–202. Springer, 2008.

[3] Patrick Eugster and K R Jayaram. EventJava: An Extension of Java for Event Correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] G Stewart Von Itzstein and David A Kearney. The Expression of Common Concurrency Patterns in Join Java. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Volume 2*, pages 1021–1021, Las Vegas, Nevada, USA, June 2004.

[5] K R Jayaram and Patrick Eugster. Scalable Efficient Composite Event Detection. In Dave Clarke and Gul Agha, editors, *12th International Conference on Coordination Models and Languages (COORDINATION 2010)*, pages 168–182, Amsterdam, The Netherlands, June 2010.

[6] Luis daniel Benavides Navarro, Andrés Barrera, Kiyoshige Garcés, and Hugo Arboleda. Detecting and Coordinating Complex Patterns of Distributed Events with KETAL. *Electr. Notes Theor. Comput. Sci.*, 281:127–141, 2011.

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

[8] Friedman Mattern. Virtual Time and Global states of Distributed Systems. In *Proceedings of the international Workshop on Parallel and distributed Algorithms*, Chateau de Bonas, France, October 1988.

[9] Anders M ller. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. Technical report, 2010.

[10] Bela Ban. JGroups, A Toolkit for Reliable Multicast Communication. 2002.