FAIDECS: Fair Decentralized Event Correlation

Gregory Wilkin, K. Jayaram, Patrick Eugster, Ankur Khetrapal

FAIDECS: Fair Decentralized Event Correlation*

Gregory Aaron Wilkin, K.R.Jayaram, Patrick Eugster, and Ankur Khetrapal

Department of Computer Science
Purdue University
{gwilkin,jayaram,peugster}@cs.purdue.edu,ankur@alumni.purdue.edu

Abstract. Many distributed applications rely on *event correlation*. Such applications, when not built as ad-hoc solutions, typically rely on centralized correlators or on *broker overlay* networks. Centralized correlators constitute performance bottlenecks and single points of failure; straightforwardly duplicating them can hamper performance and cause processes interested in the same correlations to reach different outcomes. The latter problem can manifest also if broker overlays provide redundant paths to tolerate broker failures as events do not necessarily reach all processes via the same path and thus in the same order.
This paper describes FAIDECS, a generic middleware system for *fair* decentralized correlation of events *multicast* among processes: processes with identical interests reach identical outcomes, and subsumption relationships among subscriptions are considered for respectively delivered *composite* events. Based on a generic subset of FAIDECS's predicate language, we introduce properties for composite event deliveries in the presence of process failures and present novel decentralized algorithms implementing these properties. Our algorithms are compared under various workloads to solutions providing equivalent guarantees.

Keywords: event, correlation, fair, reliable, multicast, decentralized

1 Introduction

The abstraction of application *events* is useful not only for reasoning about distributed systems [18], but also for building such systems [5,26].

Events: composition and correlation. Event *correlation* [8] enables higher-level reasoning about interactions by supporting the assembly of *composite* events from elementary events [20,19]. Traditional uses of correlation include intrusion detection [17]; network monitoring [16] enables the improvement of resource usage, e.g., in data centers. More recent application scenarios for correlation include embedded and pervasive systems [13], and sensor networks [22]. *Complex event processing* (CEP) is a computing paradigm based on event correlation, with applications to business process management and algorithmic trading.

* Financial support by NSF grants 0644013 and 0834529, DARPA grant N11AP20014. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF or DARPA.

Challenges for event correlation middleware. Reasoning about event composition is, however, involving. Early work in active databases [6] explored syntax and semantics of correlation, pinpointing options. Consider a sequence of events $e_1^1 \cdot e_2^1 \cdot e_1^2$, where e_l^k is a the l-th received event (instance) of event type T_k. This sequence can be matched by a "subscription" correlating two event types T_1 and T_2 as $[e_1^1, e_1^2]$ (*first received* first) or as $[e_2^1, e_1^2]$ (*most recent* first). However, corresponding systems are centralized and consider events to be *uni*cast.

Many theoretical and practical efforts on event correlation in publish/subscribe systems [5] consider decentralized setups and *multi*cast but focus on efficiency or the number of aggregations, yielding only best-effort guarantees on event delivery. Consider an online auction where the bidding price of a product or advertisement slot is event-driven. If two processes participating in the auction observe the same events in different orders (e.g., one receives the sequence above, the second one receiving $e_1^2 \cdot e_2^1 \cdot e_1^1$), then the event correlation middleware might be unfair to the first process if e_1^2 has information that is critical to placing an optimal bid. Or, consider assembly line surveillance through two monitors for fault tolerance. If they observe events differently, they might yield contradicting reports or alarms. During decentralized event correlation, one might not only expect that processes with identical subscriptions deliver identical sets of events, but also that if the subscription of a first process p_i "covers" that of a second process p_j, then p_i would deliver anything that p_j does. In production chains, the same complex events triggering alarms can be combined with further events for taking actions further down the chain or triggering more specific alarms. Such *subsumption* is natural in publish/subscribe systems and even key to scalability [5]. Of course, correlation-based systems can currently be designed individually to achieve such properties, e.g., by using proxy processes to merge and multiplex event streams to replicas to achieve agreement; corresponding solutions are hardly generic though, and can introduce bottlenecks to performance and dependability.

Contributions. This paper presents FAIDECS (*FAI*r *D*ecentralized *E*vent *C*orrelation *S*ystem – "Fedex"), a middleware system for *fair* decentralized correlation of events *multicast* among processes. Our exact contributions are:

- After presenting related work (Section 2) and introducing the system model and assumptions (Section 3), we present clear and feasible *properties* for *aggregated* deliveries of *sets* of events based on a concise and generic event correlation sub-grammar in FAIDECS (Section 4). While in single event (message) delivery scenarios, several families of properties have been proposed and investigated (e.g., agreed delivery [14], probabilistic delivery [4], ordering properties [11]), corresponding properties for the better understanding of correlation-based systems and ensuring "*logical correctness and integrity*" [21] are namely still lacking. Our properties provide *fairness* in the face of failures of processes responsible for merging events: either all or none of the depending processes cease to receive the desired events, while common overlays (e.g., [19]) might continue to deliver differing sets of events

to subsets of interested processes. Our properties also include a notion of subsumption on correlation patterns.
- We introduce novel pragmatic algorithms implementing our delivery properties (Section 5). For illustration purposes, we first describe simple algorithms based on a group broadcast black box. Then we present decentralized solutions implemented in FAIDECS based on a distributed hash-table (DHT), and present the use of lightweight redundancy mechanisms used for fault tolerance.
- An implementation of our algorithms in FAIDECS is evaluated under different workloads (Section 6). We quantify the benefits of our decentralized approach by comparing them with sequencer-based and token-based total order broadcast protocols providing comparable properties.

We conclude with final remarks in Section 7. Due to space limitations, we refer to a companion technical report [24] for discussions of alternative properties, or a formal proof that agreement on *composite* events *requires* a total order on individual events or an equivalent oracle.

2 Related Work

Many early approaches for composite event detection are based on *active databases* that employ *centralized* detection of events (e.g., [6]). A composite event is a pattern of events that a subscriber may be interested in. A composite *subscription* is a pattern describing the interests of the subscriber.

Event correlation has been vigorously investigated in the context of *content-based publish/subscribe systems*. Most such systems rely on a *broker network* for routing events to the subscribers (e.g., SIENA [5] and Gryphon [2]). *Advertisements* are typically used to form routing trees in order to avoid propagating subscriptions by flooding the broker network. Upon receiving an event e, a broker determines the subset of parties (subscribers and brokers) with matching interests, and forwards e to them. Subscription *subsumption* [5] is used to summarize subscriptions and avoid redundant matching on brokers and redundant traffic among them. If any event e that matches a first subscription also matches a second one, then the latter subscription subsumes the former one.

A broker network can be used to gather all publications for the elementary subscriptions and perform correlation matching. A successful match yields a composite event which is delivered to interested subscribers, where no guarantees are typically provided on correlation. If the events matching a composite subscription shared by two subscribers are produced by several publishers, then unless the subscribers are connected to a same edge broker, they may receive the events through different routes. This leads to different orders among the events and consequently to different composite events for the two subscribers. PADRES [19] performs composite event detection for each subscription at the first broker that accumulates all the individual subscriptions, providing no global properties. In the context of Hermes [20], *complex event detectors* using an interval timestamp model are proposed as a generic extension for existing middleware

architectures. Hermes uses a DHT to determine rendezvous nodes for publishers and subscribers; however, this can create single points of failure. The framework we propose is inspired by Hermes in that our framework uses specific merger nodes for specific combinations of types, determined by a DHT. However, we replicate the mergers for availability and connect them such as to ensure agreement, ordering and subsumption on composite events.

Stream processing is a paradigm closely related to event correlation and much investigated in the last few years. Research around database-backed systems like Aurora [1] or Borealis [23] has led the path. These systems, however, focus on correlation over streams of events with respect to single destinations and do not consider multicasting. Straightforwardly merging two same streams at two different nodes leads to different outcomes. StreamBase[1] is a commercial offspring of these efforts. Cayuga [8] is a generic correlation engine supporting correlation across streams and is based on a very expressive language but is centralized. The Gryphon publish/subscribe systems has similarly added support for streams [26]. Again, the focus is efficiency, leaving properties unclear.

3 Preliminaries

We assume a system Π of *processes*, Π=$\{p_1, ..., p_u\}$ connected pairwise by reliable channels [3] offering primitives to SEND (non-blocking) and receive (RECEIVE) messages. We consider a crash-stop failure model [14], i.e., a *faulty* process may stop prematurely and does not recover. We assume the existence of a discrete global clock to which processes do not have access and that an algorithm run R consists in a sequence of events on processes. That is, one process performs an action per clock tick which is either of a (a) protocol action (e.g., RECEIVE), (b) an internal action, or (c) a "no-op". A process is faulty in a run R if it fails during R, otherwise *correct*.

A failure pattern F is a function mapping clock times to processes, where $F(t)$ gives all the crashed processes at time t. Let $crashed(F)$ be the set of all processes $\in \Pi$ that have crashed during R. Thus, for a correct process p_i, $p_i \in correct(F)$ where $correct(F) = \Pi - crashed(F)$ [14].

For brevity and clarity, we adopt in the following a more formal notation for properties than common. Consider, for instance, the well-known problem of Total Order Broadcast (TOBcast) [14] defined over primitives TO-BROADCAST and TO-DELIVER, which will be used for comparison later on. We denote $\text{TO-DELIVER}^i(e)_t$ as the TO-delivery of a message conveying an event e by process p_i at time t, and similarly, $\text{TO-BROADCAST}^i(e)_t$ denotes the TO-broadcasting of e by p_i at time t. We elide any of i, t, or e when not germane to the context. We write $\exists a$ for an action a (e.g., SEND, TO-BROADCAST) as a shorthand for $\exists a \in R$. The specification of Uniform TOBcast thus becomes:

TOB-No Duplication: $\exists \text{TO-DELIVER}^i(e)_t \Rightarrow \nexists \text{TO-DELIVER}^i(e)_{t'} \mid t' \neq t$

TOB-No Creation: $\exists \text{TO-DELIVER}(e)_t \Rightarrow \exists \text{TO-BROADCAST}(e)_{t'} \mid t' < t$

[1] http://www.streambase.com/.

TOB-Validity: $\exists$to-broadcast$^i(e) \ \wedge \ p_i \in \mathit{correct}(F) \Rightarrow \exists$to-deliver$^i(e)$

TOB-Agreement: $\exists$to-deliver$^i(e) \Rightarrow \forall p_j \in \mathit{correct}(F) \backslash \{p_i\}, \exists$to-deliver$^j(e)$

TOB-Total Order: $\exists$to-deliver$^i(e)_{t_i}$, to-deliver$^i(e')_{t'_i}$, to-deliver$^j(e)_{t_j}$, to-deliver$^j(e')_{t'_j} \ \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

4 FAIDECS Model

In this section, we specify composite events in FAIDECS and the properties achieved for corresponding deliveries (DELIVER) with respect to individually generated (MULTICAST) events. In contrast to traditional settings, DELIVER is parameterized by a "subscription" Φ and delivers *ordered sets* of typed messages representing events.

4.1 Predicate Grammar

Sets of delivered events — *relations* — represent events aggregated according to specific subscriptions. Subscriptions are combinations of predicates on events in disjunctive normal form based on the following grammar (extended BNF):

Disjunction $\Psi ::= \Phi \mid \Phi \vee \Psi$	*Operation* $op ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq$
Conjunction $\Phi ::= \rho \mid \rho \wedge \Phi$	*Predicate* $\rho ::= T[i].a \ op \ v \mid T[i].a \ op \ T[i].a \mid T[i] \mid \top$

$T[i].a$ denotes an attribute a of the i-th *instance* of type T ($T[i]$) and v is a value. As syntactic sugar, we can allow predicates to refer to just T, which can be automatically translated to $T[1]$. We may use this in examples for simplicity. A type T is characterized by an ordered set of attributes $[a_1, ..., a_n]$, each of which has a type of its own – typically a scalar type such as `Integer` or `Float`. An event e of type T is an ordered set of values $[v_1, ..., v_n]$ corresponding to the respective attributes of T. We assume that types of values in predicates conform with the types of events (e.g., through static type-checking [9]). $T(e)$ returns the type of a given event e. It is important to note that we do *not* introduce a set of uniquely identified types $\{T_1, ..., T_w\}$ as we do for processes. This keeps notation more brief in that we can use $[T_1, ..., T_k]$ to refer to an arbitrary ordered set of k types, as opposed to something of the form $[T_{j_1}, ..., T_{j_k}]$.

To later simplify properties, we introduce the *empty* predicate $\top$, which trivially yields *true*. A predicate that compares a single event attribute to a value or two event attributes on the *same* event, i.e., on the same instance of a same type (e.g., $T_k[i].a \ op \ T_k[i].a'$), is a *unary* predicate. When two distinct events (two distinct types or different instances of the same type) are involved, we speak of *binary* predicates ($T_k[i].a \ op \ T_l[j].a', \ k \neq l \vee i \neq j$). We also allow *wildcard* predicates of the form $T[i]$ to be specified; such predicates simply specify a desired type $T[i]$ of events of interest. $T[i]$ implicitly also declares $T[k] \ \forall k \in [1..i-1]$ if not already explicitly declared as part of other predicates in the subscription.

We assume, for presentation brevity, a single subscription per process. The disjunction representing process p_i's subscription is represented as $\Psi(p_i)$. We

also rule out disjunctions with several identical conjunctions. In practice, we can simply remove all but one copy. By abuse of notation but unambiguously, we sometimes handle disjunctions (or conjunctions) as sets of conjunctions (or predicates). We write, for instance, $\rho_l \in \Phi \Leftrightarrow \Phi = \rho_1 \wedge ... \wedge \rho_k$ with $l \in [1..k]$.

For the following, consider an example subscription Ψ_S for an increase in three successive stock quotes after a quarterly earnings report:

$\Psi_S =$ `StockQuote[0].time` $>$ `EarningsReport[0].time` $\wedge$
`StockQuote[1].value` $>$ `StockQuote[0].value` $\wedge$
`StockQuote[2].value` $>$ `StockQuote[1].value`

We would probably want to introduce arithmetic operators on values [15] to express, e.g., that the local publication `time` of the first stock quote is within some interval of that of the earnings report. Our grammar can be easily extended by such *deterministic* constructs but is intentionally kept simple for presentation and to illustrate the independence of our algorithms from specific grammars.

4.2 Predicate Types and Evaluation

We assume that a deterministic order $\prec$ exists within subscriptions based on the names of event types, attributes, etc., which can be used for re-ordering predicates within and across conjunctions. This ordering can be lexical or based on priorities on event types and is necessary for even simplest forms of determinism and agreement. We consider subscriptions to be already ordered accordingly.

The number of events involved in a subscription is given by the number of its types and corresponding instances. More precisely, the types involved in a subscription are represented as *sequences* as they are ordered, and the same type can be admitted multiple times. Such sequences can be viewed as the *signatures* of predicates, defined as follows:

$$\begin{array}{llll} \mathbb{T}(\Phi \vee \Psi) & = \mathbb{T}(\Phi) \uplus \mathbb{T}(\Psi) & \mathbb{T}(T[i].a\ op\ v) & = \mathbb{T}(T[i]) \\ \mathbb{T}(\rho \wedge \Phi) & = \mathbb{T}(\rho) \uplus \mathbb{T}(\Phi) & \mathbb{T}(\top) & = \emptyset \\ \mathbb{T}(T_1[i].a_1\ op\ T_2[j].a_2) & = \mathbb{T}(T_1[i]) \uplus \mathbb{T}(T_2[j]) & \mathbb{T}(T[i]) & = [\underbrace{T, ..., T}_{i\times}] \end{array}$$

$\uplus$ stands for in-order union of sequences defined below:

$$\emptyset \uplus [T, ...] = [T, ...] \qquad [T, ...] \uplus \emptyset = [T, ...]$$

$$[\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...] = \begin{cases} [\underbrace{T_1, ..., T_1}_{i\times}] \oplus ([T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...]) & T_1 \prec T_2 \\ [\underbrace{T_2, ..., T_2}_{j\times}] \oplus ([T_2', ...] \uplus [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...]) & T_2 \prec T_1 \\ [\underbrace{T_1, ..., T_1}_{max(i,j)\times}] \oplus ([T_1', ...] \uplus [T_2', ...]) & T_1 = T_2 \end{cases}$$

Above, $\oplus$ represents simple concatenation. In the previous example, the types involved are thus [`EarningsReport`, `StockQuote`, `StockQuote`, `StockQuote`].

Any subscription Ψ thus involves a sequence of event types $\mathbb{T}(\Psi)$=$[T_1, ..., T_n]$, where we can have for $i, j \in [1..n], i < j$ such that $\forall k \in [i..j], T_k = T_i = T_j$. That

is, we can have subsequences of identical types. Such a subsequence represents a stream of events of the respective type of length $j-i+1$ ($T_k[1], ..., T_k[j-i+1]$).

A subscription is correspondingly evaluated for an ordered set of events $[e_1, ..., e_n]$, where e_i is of type T_i. The evaluation of a conjunction Φ on a relation is written as $\Phi[e_1, ..., e_n]$. For evaluation of an attribute a on an event e_i, we write $e_i.a$. Evaluation semantics for predicates are defined as follows:

$$(\Phi \vee \Psi)[e_1, ..., e_n] = \Phi[e_1, ..., e_n] \vee \Psi[e_1, ..., e_n] \qquad (T)[e_1, ..., e_n] = \mathit{true}$$
$$(\rho \wedge \Phi)[e_1, ..., e_n] = \rho[e_1, ..., e_n] \wedge \Phi[e_1, ..., e_n] \qquad (\top)[e_1, ..., e_n] = \mathit{true}$$

$$(T[i].a\ op\ v)\ [e_1, ..., e_n] = \begin{cases} e_{k+i-1}.a\ op\ v & T(e_k) = T \wedge (T(e_{k-1}) \neq T \\ & \quad \vee (k-1) = 0) \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$(T_1[i].a_1\ op\ T_2[j].a_2)\ [e_1, ..., e_n] = \begin{cases} e_{k+i-1}.a_1\ op\ e_{l+j-1}.a_2 & T(e_k) = T_1 \wedge (T(e_{k-1}) \neq T_1 \\ & \quad \vee (k-1) = 0) \wedge T(e_l) = T_2 \\ & \quad \wedge (T(e_{l-1}) \neq T_2 \vee (l-1) = 0) \\ \mathit{false} & \text{otherwise} \end{cases}$$

For brevity we may write simply $\Phi[...]$ for $\Phi[...] = \mathit{true}$.

A process p_i delivers events in response to its subscription $\Psi(p_i)$ through DELIVER. We consider this primitive to be generically typed, i.e., we write DELIVER$_\Phi([e_1, ..., e_n])$ to deliver a relation $[e_1, ..., e_n]$, where e_j is of type T_j such that $\mathbb{T}(\Phi)=[T_1, ..., T_n]$. DELIVER$^i_\Phi([e_1, ..., e_n])_t$ denotes a delivery on process p_i in response to Φ at time t, and MULTICAST$^i(e)_t$ defines the multicast of an event e by p_i at time t. Again i, t, etc. may be omitted when not germane to the context.

4.3 Properties

We now present properties for composite events in FAIDECS defined over primitives MULTICAST and DELIVER. From here on, *deliver* refers to DELIVER (vs. *TO-deliver* for TO-DELIVER), and *multicast* refers to MULTICAST (vs. *TO-broadcast*). See [24] for detailed discussions of alternative properties.

Basic safety properties. The basic safety properties for FAIDECS are MDM-NO DUPLICATION, MDM-NO CREATION and ADMISSION as shown below:

MDM-NO DUPLICATION: $\exists \text{DELIVER}^i_\Phi([..., e, ...])_t \Rightarrow \nexists \text{DELIVER}^i_\Phi([..., e, ...])_{t'} \mid t' \neq t$

MDM-NO CREATION: $\exists \text{DELIVER}_\Phi([..., e, ...])_t \Rightarrow \exists \text{MULTICAST}(e)_{t'} \mid t' < t$

ADMISSION: $\exists \text{DELIVER}^i_\Phi([e_1, ..., e_n]) \mid \mathbb{T}(\Phi) = [T_1, ..., T_n] \Rightarrow \Phi \in \Psi(p_i) \wedge \Phi[e_1, ..., e_n] \wedge \forall k \in [1..n] : T(e_k) = T_k$

The MDM-NO DUPLICATION property implies that a same event is delivered at most once for a given conjunction, which may be opposed to certain systems that allow a same event to be correlated multiple times. Our property could easily be substituted to allow a delivery for *every instance* of a type in a given conjunction. We omit this for simplicity of the presented properties and algorithms. MDM-NO CREATION is similar to TO-broadcast specifications [14]

in that an event may only be delivered if multicast. ADMISSION ensures type safety and that all events in a relation match the subscription.

Liveness. ADMISSION can trivially hold while not delivering anything. We have to be careful about providing strong delivery properties on *individually* multicast events though, as events may depend on others to match a given conjunction. Nonetheless, we want to rule out bogus implementations which simply discard all events. We thus propose the following complementary liveness properties:

CONJUNCTION VALIDITY: $\exists \text{MULTICAST}(e_l^k), k \in [1..n], l \in [1..\infty] \ \wedge \ p_i \in \mathit{correct}(F) \ \wedge \ \exists \Phi \in \Psi(p_i) \mid \Phi[e_l^1, ..., e_l^n] \ \Rightarrow \exists \text{DELIVER}_{\Phi}^{i}([...])_{t_j} \mid j \in [1..\infty]$

EVENT VALIDITY: $\exists \text{MULTICAST}^{i}(e^x), \text{MULTICAST}^{k,l}(e_l^k), k \in [1..n] \backslash x, \ l \in [1..\infty]$
$\{p_i, p_j, p_{k,l}\} \subseteq \mathit{correct}(F) \mid \Phi \in \Psi(p_j) \wedge \mathbb{T}(\Phi) = [T_1, ..., T_n] \ \wedge \ \forall z \in [w..y],$
$T_z = T(e^x) \wedge \nexists (T(e^x)[x-w+1].a_1 \ op \ T[r].a_2) \in \Phi \mid (T \neq T(e^x) \vee r \neq x-w+1) \wedge$
$\Phi[e_l^1, ..., e_l^{x-1}, e^x, e_l^{x+1}, ..., e_l^n] \Rightarrow \exists \text{DELIVER}_{\Phi}^{j}([..., e^x, ...])$

These two properties handle the two possible cases that can arise. The first property deals with dependencies across events and can be paraphrased as follows: "If for a correct process p_i, there is an infinite number of relations of matching events that are successfully multicast, then p_i will deliver infinitely many such relations." This property is reminiscent of the FINITE LOSSES property of fair-lossy channels [3]. It allows matching algorithms to discard *some* events for practical purposes such as agreement and ordering, yet ensures that when matching events are continuously multicast, a corresponding process will continuously deliver. From the example presented in Section 4.1, as long as events of both types are inifinitely published such that infinitely often, three successive, increasing stock quotes are multicast after an earnings report, there will be an infinite number of delivered relations.

EVENT VALIDITY provides a property analogous to validity for single-message deliveries (e.g., TOBcast): If an event is multicast by a correct process p_i, and its delivery in response to a conjunction on some correct process p_j is *not* conditioned by binary predicates with other event types, then the event must be delivered by p_j if matching events of all other types are continuously multicast. This latter condition is necessary because the delivery of the event, even in the absence of binary predicates, requires the *existence* of other events (by nature of correlation). The condition also ensures that any unary predicates on the respective event type are satisfied. Note that in the case of multiple instances of that type, for each of which there are only unary predicates that match, the property does not force an event to be delivered more than once as the position of the event is not fixed in the implied delivery. The example in Section 4.1 does not present a unary predicate, and thus would not be affected by this property. If the subscription Ψ_S were extended to trigger only if the value of the U.S. dollar is below some value `v` as in $\Psi'_S = \Psi_S \ \wedge$ `USDollar.value < v`, then any event matching this predicate will be delivered with the entire relation given by Ψ_S.

Note also that none of these properties is impacted by the presence of multiple instances of a same type in a conjunction. An infinite flow of events of some type implies a multiple (a finite number) of infinite flows of that type.

Agreement. The properties so far ensure that as long as matching events are being multicast, processes will eventually deliver relations. We are, however, interested in stronger properties for these delivered relations, which ensure fairness for relations delivered across processes. We define Covering Agreement:

Covering Agreement: $\exists \text{deliver}^{i}_{\Phi\wedge\Phi'}([e_1,...,e_n,...]) \mid ((\mathbb{T}(\Phi)=[T_1,...,T_n])\cap\mathbb{T}(\Phi')) = \emptyset \Rightarrow \forall p_j \in correct(F)\backslash\{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{deliver}^{j}_{\Phi}([e_1,...,e_n])$

Subsumption only allows "extending conjunctions to the right" as determinism requires some given order for matching. Intuitively, subsumption in the presence of binary predicates is limited since, when comparing two subscriptions with same types, an event of a first type might match both subscriptions without implying that the same holds for a second event.

Note that Covering Agreement is not defined in a symmetric way (with $\Phi \wedge \Phi'' \in \Psi(p_j)$), as the presence of a matching set of events for a conjunction Φ' does not imply a timely or even eventual occurrence of a matching set for another sub-relation Φ'' conjoined by p_j with Φ.

Thus, the example subscriptions Ψ_S, as defined in Section 4.1, and Ψ'_S, defined in 4.3, would exhibit the necessary conditions for Covering Agreement. That is, the common predicates over the `EarningsReport` and `StockQuote` types would yield the same (sub)-relations for Ψ_S and Ψ'_S, where Ψ'_S would deliver relations containing the above with an additional event of type `USDollar`.

4.4 Total Order

Intuitively, and as we will illustrate in the following sections, a total order on individual events can be used to achieve agreement on relations. In fact, it is necessary to do so (see [24] for a formal proof). On the upside, this can be exploited to provide corresponding relation-level properties. We define three types of total order properties below:

Event Total Order: $\exists \text{deliver}^{i}_{\Phi}([...,e,...])_{t_i}, \text{deliver}^{i}_{\Phi}([...,e',...])_{t'_i}, \text{deliver}^{j}_{\Phi'}([...,e,...])_{t_j}, \text{deliver}^{j}_{\Phi'}([...,e',...])_{t'_j} \mid T(e)=T(e') \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

Conjunction Total Order: $\exists \text{deliver}^{i}_{\Phi\wedge\Phi'}([e_1,...,e_n,...])_{t_i}, \text{deliver}^{i}_{\Phi\wedge\Phi'}([e'_1,...,e'_n,...])_{t'_i}, \text{deliver}^{j}_{\Phi\wedge\Phi''}([e_1,...,e_n,...])_{t_j}, \text{deliver}^{j}_{\Phi\wedge\Phi''}([e'_1,...,e'_n,...])_{t'_j} \mid ((\mathbb{T}(\Phi)=[T_1,...,T_n])\cap\mathbb{T}(\Phi')) = \emptyset \wedge (\mathbb{T}(\Phi)\cap\mathbb{T}(\Phi'')) = \emptyset \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

Disjunction Total Order: $\exists \text{deliver}^{i}_{\Phi}([e_1,...,e_n])_{t_i}, \text{deliver}^{i}_{\Phi'}([e'_1,...,e'_m])_{t'_i}, \text{deliver}^{j}_{\Phi}([e_1,...,e_n])_{t_j}, \text{deliver}^{j}_{\Phi'}([e'_1,...,e'_m])_{t'_j} \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

None of the properties includes any of the others. Event Total Order ensures that there is a total (sub-)order on the events of a same type. Conjunction Total Order ensures that (sub-)*relations* delivered to identical (sub-)conjunctions are delivered in a total order. An implementation which *never* enforces Conjunction Total Order, i.e., delivers no two same relations on two processes with identical (sub-)conjunctions, could still ensure Event Total Order. Perhaps more obvious is that, inversely, Event Total Order does not

imply Conjunction Total Order. Disjunction Total Order further sets our model apart from many single-event delivery multicast settings (e.g., traditional publish/subscribe), where subscriptions are conjunctions, and disjunctions are viewed as being expressed independently through multiple conjunctions. Our property strives for total order *across* relations delivered to *distinct* conjunctions in a *same* disjunction.

5 Algorithms

We now present ways to implement the properties proposed in the previous section. For illustration purposes, we first outline an approach relying straightforwardly on a total order across multicast events of *all* types. Then, we present novel decentralized algorithms achieving the same properties, leveraging our notion of subscription subsumption.

5.1 Total Order Broadcast Black Box

A straightforward solution for deterministic event correlation across all processes is to rely on a Total Order Broadcast "black box," with primitives TO-BROADCAST and TO-DELIVER for individual events, ensuring that all correct processes eventually TO-deliver all TO-broadcast events in the same order. To multicast an event e of any type, a process simply performs TO-BROADCAST(e); a TO-DELIVER(e) is handled in a deterministic manner described shortly. Many implementations exist, tolerating different failure patterns [7].

Conjunctions. For simplicity, we first focus on single conjunctions for the algorithm in Figure 1 before expounding on generic disjunctions. That is, subscription Ψ_i of process p_i consists in a single conjunction Φ_i. Disjunction Total Order, in this case, becomes subsumed by Conjunction Total Order.

The algorithm in Figure 1 uses *first received* matching semantics and *prefix+infix* disposal. In short, the former means that events are matched on a process in the order received by that process. The latter implies the following: Upon a successful match $[e_1, \ldots, e_n]$, for each event e_i, all events of the same type received prior to e_i are discarded via the garbage collection mechanism DEQUEUE. These semantics are further elaborated on below.

Each process p_i maintains one queue Q per event type in its conjunction $\Phi{=}\Psi(p_i)$. For example, for a conjunction $\Phi = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_2$ and $\rho_2 = T_1.a_1 < 20$, the subscriber maintains one queue for events of type T_1 and one for events of type T_2. When TO-delivering an event, p_i will loop *once* by line 20 and first checks whether the type of the event is in p_i's subscription. If so, p_i attempts to ENQUEUE the event. $Q[T(e)] \oplus e$ denotes the appending of event e to the queue of type $T(e)$. The ENQUEUE primitive returns *true* if the event has been ENQUEUEd, which means that it satisfies all unary predicates on the respective types in the conjunction. Then p_i proceeds to MATCHing. Any single received event may complete up to one relation. If a match $[e_1, \ldots, e_n]$

Executed by every process p_i

```
1: init
2:     Ψ ← Φ_1 ∨ ... ∨ Φ_o
3:     Φ_l ← ρ_1 ∧ ... ∧ ρ_m
4:     Q_l[T] ← ∅
5: To MULTICAST(e):
6:     TO-BROADCAST(e)
7: function MATCH ([e'_1, ..., e'_n], Φ, Q)
8:     T ← T_{n+1} | T(Φ) = [T_1, ..., T_{n+1}, ...]
9:     l ← max(j | Q[T] = e_1 ⊕ ... ⊕ e_j ⊕ ... ⊕ e_h) |
         ∃k ∈ [1..n] : e_j = e'_k
10:    for all k = (l + 1)..h do
11:        if |T(Φ)| = n + 1 then
12:            if Φ[e'_1, ..., e'_n, e_k] then
13:                return [e'_1, ..., e'_n, e_k]
14:        else
15:            E ← MATCH([e'_1, ..., e'_n, e_k], Φ, Q)
16:            if E ≠ ∅ then
17:                return E
18:    return ∅
19: upon TO-DELIVER(e) do
20:    for all Φ_l ∈ Ψ | T(e) ∈ T(Φ_l) in order do
21:        if ENQUEUE(e, Φ_l, Q_l) then
22:            [e_1, ..., e_k] ← MATCH(∅, Φ_l, Q_l)
23:            if k ≠ 0 then
24:                DEQUEUE([e_1, ..., e_k], Q_l)
25:                DELIVER_{Φ_l}([e_1, ..., e_k])
26: function ENQUEUE (e, Φ, Q)
27:    win ← max(j | ∃...T(e)[j].a... ∈ Φ)
28:    if ∀j = 1..win ((∃ρ = (T(e)[j].a op v) ∈
         Φ | ¬ρ[e]) ∨ (∃(ρ = T(e)[j].a op
         T(e)[j].a') ∈ Φ | ¬ρ[e])) then
29:        return false
30:    else
31:        Q[T(e)] ← Q[T(e)] ⊕ e
32:        return true
33: procedure DEQUEUE([e_1, ..., e_m], Q)
34:    for all Q[T] = ... ⊕ e_k ⊕ e ⊕ ..., k ∈ [1..m]
       do
35:        Q[T] ← e ⊕ ...
```

Fig. 1: Conjunctions/disjunctions with Total Order Broadcast.

is identified, the corresponding events are discarded (DEQUEUE) and for each event e_i, all preceding events of the same type are discarded from the respective queue for that type. MATCH iterates through the queues deterministically. The semantics attempt to find the *first* instance of the first type in Φ for which there are events of the remaining types with which Φ is satisfied. Among all such possibilities, the algorithm recursively seeks for a match with the *first* instance of the second type in Φ, etc. until a match is found or all possibilities are exhausted. For multiple instances of a same type, a first instance is recursively matched with the *first follow-up instance* in the same queue until the needed number of instances is found for that type or the queue is exhausted.

Assuming that the underlying TOBcast primitive ensures TOB-No Creation and TOB-No Duplication (see Section 3), it is easy to see how the algorithm of Figure 1 ensures the corresponding MDM-No Creation and MDM-No Duplication properties defined in Section 4.3. An event e, matching all unary predicates of a conjunction Φ, is successfully added to the corresponding queue $Q[T(e)]$ in ENQUEUE (line 31, Figure 1). The only way in which e can be removed (and delivered) is together with a matching set of other events fulfilling Φ (line 23, Figure 1), thus ensuring Admission. If matching sets of such events are continuously TO-broadcast, then a match will eventually be determined at line 12 thus ensuring Event Validity. Conjunction Validity holds by a similar line of reasoning. The first matching, together with prefix+infix disposal, and the independent handling of events of distinct types ensures Event Total Order. If two processes p_i and p_j define conjunctions $\Phi \wedge \Phi'$ and Φ respectively, as long as Φ and Φ' are type-disjoint, then events that match with Φ are independent of any events that match with Φ'. Thus, if there is a matching relation for p_i, there is a subset of the relation for which Φ is true. Since garbage collection is deterministic and is triggered every time an event of a type in $\mathbb{T}(\Phi)$ is TO-delivered and in the same order on p_i and p_j with respect to those de-

liveries, p_i and p_j will handle respective events identically, ensuring COVERING AGREEMENT. Similarly, CONJUNCTION TOTAL ORDER holds as all processes TO-deliver all relevant events. When p_i identifies a match for $\Phi \wedge \Phi'$, with Φ and Φ' type-disjoint, p_j will have TO-delivered the respective subset of events in Φ already in the same sub-order and thus DELIVERs the respective sub-relations in the same order with any events identified for a Φ'' type-disjoint with Φ.

Disjunctions. When the subscription is a disjunction of several conjunctions, a process maintains one event queue per event type *per* conjunction. For example, for a disjunction $\Psi = \Phi_1 \vee \Phi_2$ where $\mathbb{T}(\Phi_1)=\mathbb{T}(\Phi_2)=[T_1, T_2]$, a process maintains two queues for type T_1 and then two queues for type T_2, one each for Φ_1 ($Q_1[T_1]$ and $Q_1[T_2]$) and for Φ_2 ($Q_2[T_1]$ and $Q_2[T_2]$).

Figure 1 supports multiple conjunctions in a single disjunction. The primary distinction is in the response to TO-deliveries. The primitive dispatches events to conjunctions *in order* of subscriptions. In contrast to subscriptions of one conjunction, an event can lead to multiple MATCHes and DELIVERies.

Because the MATCHing is performed deterministically, as explained previously for a given conjunction, and all processes ENQUEUE the same sets of events in the same order, COVERING AGREEMENT across any two conjunctions is met for the same reasons as for single conjunctions. This property would also be met by any unordered dispatching for multiple conjunctions. The other properties established for conjunctions remain valid due to the duplication of events appearing in distinct conjunctions of a same subscription.

DISJUNCTION TOTAL ORDER is met as any p_i and p_j defining two identical separate conjunctions TO-deliver the respective events (possibly interleaved by those for other conjunctions in $\Psi(p_i)$ and $\Psi(p_j)$ respectively) in the same order. Thus, the correlation for respective relations occurs in the same order.

A simple optimization of the algorithm for subscriptions containing several conjunctions Φ_1,...,Φ_m with a common event type T, omitted for brevity, consists in sharing the queue for T across conjunctions. An event in a queue is then tagged by the index k of a conjunction Φ_k to indicate that the event has previously been used in a match and DELIVERed for Φ_k. Earlier events of that type should then also be tagged with k. Events with tags {1, ..., m} may then be discarded. Also, the portrayed matching algorithm performs an exhaustive search and is thus not efficient; however, it suffices to illustrate the relevant properties and can be represented concisely. More elaborate and efficient matching algorithms exist, which offer the same semantics. A common approach consists in storing *partial matches* in specialized data-structures to avoid matching a given event multiple times with same events (cf. [9]). In our implementation of FAIDECS and all evaluated algorithms, we make use of the Rete [10] matching algorithm.

5.2 FAIDECS Decentralized Ordered Merging

One of the simplest and most popular approaches in practice for Total Order Broadcast consists in a sequencer, which orders *all* events. As long as the sequencer remains available (e.g., through replication), the properties presented

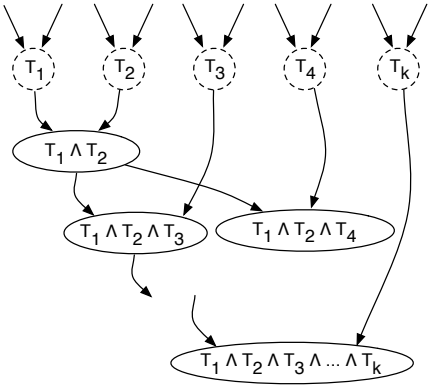


Fig. 2: $T_1 \wedge ... \wedge T_j$ denotes the *conjunction* merger for the respective types $\sqcup[T_1, ...T_j]$ (single instance per type).

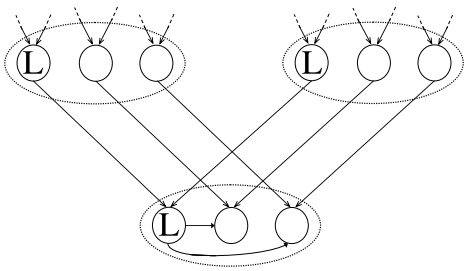


Fig. 3: Small-scale FAIDECS merger replication. Dotted ovals are "logical" mergers; circles are processes. L denotes the leader.

earlier hold under respective assumptions on failure patterns. A Consensus-based textbook Total Order Broadcast [14] yields the same properties with much better fault tolerance (typically a minority of all processes may fail), yet with a higher overhead. We now present a decentralized solution implementing the same properties, yet with much better scalability characteristics than both and inherently better fault-tolerance than a sequencer-based approach. The solution assumes a distributed hashtable (DHT) or similar mechanism for uniquely identifying a process for a given "role." Lightweight replication mechanisms used for fault-tolerance of such roles are discussed separately thereafter.

Conjunctions. We first describe an algorithm focusing on single conjunctions, providing the same properties as that of Figure 1. All processes with conjunctions on a sequence of event types $[T_1, ..., T_k]$ send their subscriptions to a same process, identified as p_j=PROCESS($\sqcup[T_1, ..., T_k]$), responsible for handling all conjunctions on the involved sequence of types *without duplicates*[2]:

$$\sqcup[T_1, ..., T_1, T_2, ...] = [T_1] \oplus \sqcup[T_2, ...]$$

The function PROCESS relies on a DHT (e.g., a deterministic lookup facility) to deterministically identify such responsible processes, called *mergers*. Lodged at the root of the thereby created overlay network (see Figure 2) are mergers responsible for individual event types T_1, T_2, etc. To ensure the properties with respect to extensions of conjunctions to the right, events undergo an *ordered merge by type* where a merger p_j=PROCESS($\sqcup[T_1, ..., T_k]$) gets events of types T_1, ..., T_k from two processes: those identified as PROCESS($\sqcup[T_1, ..., T_{k-1}]$) and PROCESS($[T_k]$). We term processes in the *role* of subscribers/publishers as *clients*.

Figure 4 presents the algorithm for merging event types and handling subscriptions corresponding to the merged types. Figure 5 presents the algorithm

[2] We could use different mergers but deduplication simplifies the algorithm.

Executed by every process p_i=PROCESS($\sqcup[T_1, ..., T_k]$)

```
1: init
2:     left ← PROCESS([⊔T_1, ..., T_{k-1}])
3:     right ← PROCESS([T_k])
4:     subs[p_j]
5:     kids[p_j]
6:     INITPARENTS()
7: procedure INITPARENTS()
8:     Ψ' ← ⋁_{Ψ∈kids∪subs} Ψ\
         {ρ ∈ Ψ | 𝕋(ρ) ∉ {[T_1], ..., [T_{k-1}]}}
9:     SEND(CON, Ψ') to left
10:    Ψ'' ← ⋁_{Ψ∈kids∪subs} Ψ\
         {ρ ∈ Ψ | 𝕋(ρ) ≠ [T_k]}
11:    SEND(CON, Ψ'') to right
12: upon RECEIVE(CON, Ψ) from p_j do
13:    kids[p_j] ← Ψ
14:    INITPARENTS()
15: upon RECEIVE(SUB, Φ) from p_j do
16:    subs[p_j] ← Φ\{ρ ∈ Φ | |𝕋(ρ)| > 1}
17:    INITPARENTS()
18: upon RECEIVE(EV, e) do
19:    for all Ψ = kids[p_j] do
20:        if ∃l, Φ ∈ Ψ | ∀ρ = T(e)[l]... ∈ Φ : ρ[e]
           then
21:            SEND(EV, e) to p_j
22:    for all Φ = subs[p_j] do
23:        if ∃l | ∀ρ = T(e)[l]... ∈ Φ : ρ[e] then
24:            SEND(EV, e) to p_j
```

Fig. 4: Ordered merging for conjunctions: mergers.

Executed by every p_i. Reuses ENQUEUE, MATCH, DEQUEUE of Figure 1

```
1: init
2:     Ψ ← Φ
3:     Φ ← ρ_1 ∧ ... ∧ ρ_m
4:     Q[T] ← ∅
5:     SEND(SUB, Φ) to PROCESS(⊔𝕋(Φ))
6: To MULTICAST(e):
7:     SEND(EV, e) to PROCESS([T(e)])
8: upon RECEIVE(EV, e) do
9:     if ENQUEUE(e, Φ, Q) then
10:        [e_1, ..., e_l] ← MATCH(∅, Φ, Q)
11:        if l > 0 then
12:            DEQUEUE([e_1, ..., e_l], Q)
13:            DELIVER_Φ([e_1, ..., e_l])
```

Fig. 5: Ordered merging for conjunctions: clients.

for client processes. Unary predicates are propagated from subscribers to mergers (line 16, Figure 4), and from mergers to their ancestor mergers in the form of disjunctions (lines 8-11) since a potential match (i.e., compliant with any unary predicates) for *any* merger or subscriber means a potential match for a parent merger. Forwarding of events received by mergers from their respective parent mergers ($left$) or processes for merged event types ($right$) happens without interruptions by other events and can be achieved by simple local synchronization.

For simplicity, the algorithm in Figure 5 handles event queues at clients. The use of shared queues on mergers as described at the end of Section 5.1, could lead to savings in global memory overhead by avoiding redundancies. In practice, we have observed that this, however, overburdens mergers, just like a propagation of complete conjunctions instead of only unary predicates to mergers.

Assuming that all subscribers are connected to mergers which are connected to each other before events are multicast, the properties described in Section 4.3 are also met by the algorithm in Figures 4 and 5 thanks to the type-ordered merging of events. COVERING AGREEMENT and CONJUNCTION TOTAL ORDER are ensured as processes with a common "prefix" in their conjunctions, which is type-disjoint with any conjoined predicates, will receive the same events for the prefix and in the same order from the corresponding conjunction merger process.

Disjunctions. For disjunctions, we essentially need to solve Total Order *Multi*-cast [12] on the event sequences output by conjunction mergers. Using time-stamps and extending the conjunction algorithm of Figures 4 and 5, order of events is established for clients as needed for disjunctions. More precisely, con-

Executed by every process p_i=PROCESS($\sqcup[T_1, ..., T_k]$). Reuses lines 1-11 of Figure 4

```
18: upon RECEIVE(EV, e)   {Rplcs lines 18-24 }
19:     for all Ψ = kids[p_j] do
20:         if ∃l, Φ ∈ Ψ | ∀ρ = T(e)[l]... ∈ Φ : ρ[e]
            then
21:             SEND(EV, e) to p_j   {end for}
22:     time ← current time {cont frm Line 21}
23:     for all   Φ = subs[p_j] do
24:         if   ∃l | ∀ρ = T(e)[l]... ∈ Φ : ρ[e] then
25:             SEND(EV, e, time) to p_j
```

Fig. 6: Disjunction-enabled ordered merging for conjunctions: mergers.

Executed by every p_i. Reuses ENQUEUE, MATCH, DEQUEUE of Figure 1

```
1: init
2:     Ψ ← Φ_1 ∨ ... ∨ Φ_o
3:     Φ_l ← ρ_1 ∧ ... ∧ ρ_m
4:     Q_l[T] ← ∅
5:     R ← ∅
6:     S[T] ← 0
7:     for all Φ_l ∈ Ψ do
8:         SEND(SUB, Φ_l) to PROCESS(⊔𝕋(Φ_l))
9: To MULTICAST(e):
10:    SEND(EV, e) to PROCESS([T(e)])
11: upon RECEIVE(EV, e, ts) do
12:    if ts > S[T(e)] then
13:        S[T(e)] ← ts
14:        R' ← {⟨e', t'⟩ ∈ R | t' < ts}
15:        R'' ← {⟨e', t'⟩ ∈ R | t' > ts}
16:        R ← R' ∪ {⟨e, ts⟩} ∪ R''
17:        for all ⟨e', t'⟩ ∈ R ordered on t' |
               t' < MIN_T(S[T]) do
18:            for all Φ_l in order do
19:                if ENQUEUE(e', Φ_l, Q_l) then
20:                    R ← R\{⟨e', t'⟩}
21:                    [e_1, ..., e_k] ← MATCH(∅, Φ_l, Q_l)
22:                    if k > 0 then
23:                        DEQUEUE([e_1, ..., e_k], Q_l)
24:                        DELIVER_{Φ_l}([e_1, ..., e_k])
```

Fig. 7: Ordered merging for conjunctions and disjunctions: clients.

junction mergers following the algorithm of Figure 6 timestamps all received messages before passing them to clients which do the actual correlation (Figure 7). There is no need for specialized disjunction mergers, which are thus omitted here for simplicity. (If using dedicated disjunction mergers, these can be arbitrarily connected among each other to cover the respective conjunctions.)

If processes send timestamps with events, to achieve order of DELIVERy for relations, an event is only ENQUEUEd (and correspondingly MATCHed) when a receiving process has received events for all other types in its subscription, and the timestamp of that event is less than all the other respective timestamps of other types. As long as all processes which are MULTICASTing events of the respective types continue to do so, for any receiving process, an event will eventually be ENQUEUEd after other events with lower timestamps of other types. This guarantees that all processes receiving the same events over a set of types will ENQUEUE and thus perform a MATCH on them one by one in the same order.

If there are any processes which multicast events at a slower rate than others, then the approach may not be as efficient with the requirement that each event of a type (before being ENQUEUEd) must wait for events of every other type with higher timestamps to be received. To solve this problem for the algorithm in Figure 7, if an event has not been received in some time interval by a *conjunction* merging process, then an "empty" event $e_\perp$ may be sent to all processes in $subs[p_j]$, indicating that pending events of other types may be respectively ENQUEUEd. Depending on the targeted scenarios (e.g., publication rate, topology) other information such as rates may be used (additionally).

MDM-No Creation and MDM-No Duplication are met as ENQUEUE and MATCH are only performed on received events, and for a given type, only events with a higher timestamp than the last event of that type are further

added to the ordered set R and queue Q_l. Since an event is never ENQUEUEd unless its type exists in the process's subscription, and MATCH is performed over every received event, ADMISSION holds. As in Section 5.2, EVENT VALIDITY and CONJUNCTION VALIDITY are retained here despite the filtering and discarding of certain events. It is easy to see that the timestamps generated by mergers follow the observed order of event reception, thus respecting CONJUNCTION TOTAL ORDER. Given that events are compared based on timestamps and merged in order of conjunctions, DISJUNCTION TOTAL ORDER is also ensured.

Joining. The algorithms presented so far *all* rely on a consistent set of event queues across all processes with the same composite subscription if any subscription is issued prior to publications. However, this consistency is violated when two such related processes subscribe to an event stream at different times with respect to the multicasting of events. In order to maintain consistency, we thus employ a simple synchronization algorithm between (a) a joining subscriber process, (b) the corresponding conjunction merger(s), and (c) one of the *existing* subscriber processes with identical conjunctions, if any. This ensures that a joining process starts with a valid state of the respective queues copied from any existing subscriber and does not miss any subsequent events from the merger received also by that existing subscriber after copying the state of its queues.

Fault tolerance. For presentation simplicity, the algorithms described thus far stipulated single processes returned by function PROCESS() as responsible for given conjunctions, which obviously provides little fault tolerance. In FAIDECS, PROCESS() returns a small fixed number of processes; i.e., the underlying DHT determines a set of replicas for such merger roles. A membership layer monitors the merger processes and ensures that their membership is consistent. Figure 3 provides an overview of the replication. A role, or "logical" merger process, is represented by 3 replicas which are contoured by a dotted line. L represents a *leader* process which determines the order between the merged types and communicates that *order* (only) to its peers. These receive the *actual events* independently as depicted in the figure. When a physical merger process (solid circles) p_i fails, its descendant(s) connect to one of p_i's peers. To ensure that no events are missed in the meantime, all replicas regularly acknowledge received and forwarded events to each other; events prior to such acknowledgements are buffered. If a process lags or fails, its peers will attempt to replace it. Using majority-based voting, a minority of (suspected) process failures can typically be tolerated at a time. In addition to benefitting fault tolerance, this small-scale replication also benefits load distribution, in that down-stream processes, including subscribers, distribute uniformly over the replicas.

6 Evaluation

To demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs, we compare a Java implementation of

FAIDECS to the algorithm of Figure 1 with 3 different JGroups-based[3] implementations for the Total Order Broadcast black box: (1) a sequencer algorithm, (2) a replicated sequencer (3 replicas) and (3) a token-based algorithm. Figure 10 summarizes our findings. An extended version of this report [25] presents further descriptions and results.

6.1 Metrics and Experimental Setup

We used two metrics – *Throughput:* the average number of events delivered per second by a subscriber, and *Latency:* the average delay between the multicasting time of an event and its delivery to a subscriber. The number of subscribers was increased from 10 to 600, and each subscriber had a randomly generated set of subscriptions. Each event consisted of 3 integer attributes with values chosen uniformly at random within [0..1000]. All processes were run on 65 nodes in a LAN. Each node is equipped with an Intel Xeon 3.2GHz dual-core processor and 2GB RAM, and runs Linux. A maximum of 15 subscriber processes were run on a single node. The maximum multicast rates varied by setup (e.g., different components became the bottleneck, selectivity of subscriptions varied). We tested scalability of FAIDECS first in terms of conjunctions and then disjunctions.

For conjunctions, we used 3 different distributions of subscriptions, which led to different workloads for actual routing and filtering of events. In scenarios A and B, we followed the setup of Figure 8, increasing the maximum number of conjoined types (and thus the depth) k from 2 to 4. For scenario A, *all* filtering occurred at end nodes rather than in mergers through the selectivity of binary predicates, which differed across conjunctions to achieve the same expected delivery rates at all subscribers in a respective level. This scenario demonstrated the limits of the overlay. In scenario B, events were filtered at the mergers through unary predicates propagated upwards from subscriptions, allowing higher aggregate multicast rates than in scenario A. Scenario C invariably had 4 event types, and subscriptions were over all 6 possible conjunctions ($\binom{4}{2}$). This allowed us to explore the potential of traffic separation. For evaluating scalability with respect to disjunctions, we used scenario D, which is the merger overlay shown in Figure 9. The maximum level was also varied (from 2 to 4). Subscribers were uniformly distributed across all merger processes and throughput/latency values were averaged for each group of subscribers for a given level.

We expect that the bottleneck in our decentralized algorithms would occur at the merger process(es) which would merge all involved types, limiting throughput consistently for all k. All values are normalized with respect to the values obtained with FAIDECS with 10 subscribers connected to a single merger for 2 types in scenario A, and with respect to the relations with the largest number of types (independent of the algorithm). Throughput here was approximately 31,400 events/s and latency 150ms. Normalization does not introduce any bias but makes comparison clear, so that values could be reported independent of subscriptions, and so that values may be reported for each level independently.

[3] `http://www.jgroups.org`

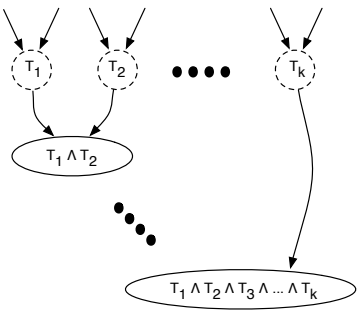


Fig. 8: Setup for conjunctions (scenarios A and B).

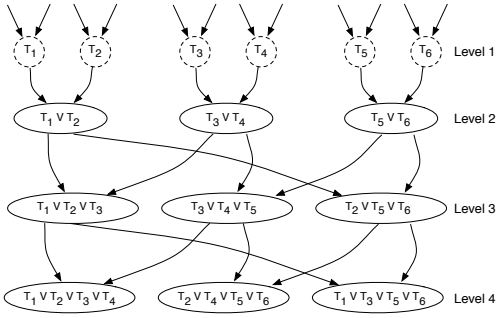
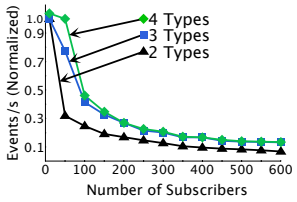
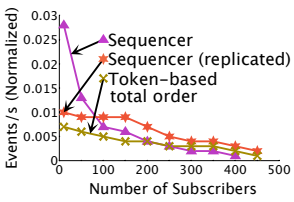


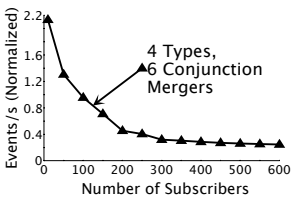
Fig. 9: Setup for disjunctions (scenario D).



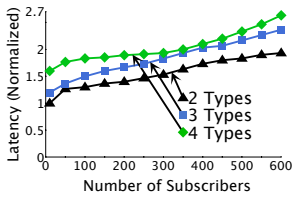
(a) Scenario A/B throughput for conjunctions.



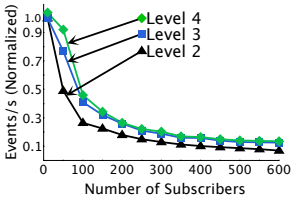
(b) Other total order implementations.



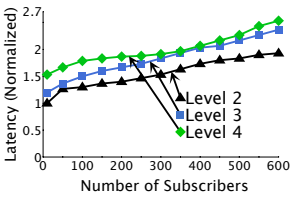
(c) Scenario C throughput for conjunctions.



(d) Scenario A latency for conjunctions.



(e) Scenario D throughput for disjunctions.



(f) Scenario D latency for disjunctions.

Fig. 10: Comparing conjunction/disjunction algorithms to a sequencer based approach.

6.2 Conjunctions

Figure 10(a) displays the trend in throughput as the system scales to more subscribers in scenarios A and B with varying number of event types/levels k (see Figure 8). FAIDECS scales very well compared to the approaches shown in Figure 10(b), shown separately for a clear relationship among the three implementations since the values start at nearly 3% (about 950 events/s) and remained consistent in all scenarios. Note that IP-multicast was turned off in the test environment which could help throughput for both FAIDECS and the JGroup implementations. In Figure 10(b), the token-based algorithm starts with a higher throughput than the sequencer-based one as there were few multicasters competing over the token, but its performance degrades faster due to the inherent cost of its high fault tolerance. Replication helps performance in both

FAIDECS and the replicated sequencer due to the load balancing of replicas of a same logical merger process, though less and with an initial cost for the replicated sequencer. The total throughput remained approximately the same in scenarios A and B since propagation of events by mergers was the bottleneck.

Figure 10(c) illustrates the scalability and the high throughput of FAIDECS when subscriber interests are in largely disjoint types, following scenario C. Thus, FAIDECS scales very well with the addition of an arbitrary number of types to a system, even with transitive correlation across them as in scenario C, given enough merger process nodes to support them – the high throughput (about double that of two types for scenario A) occurs because every merger only handles relatively few subscribers compared to the other scenarios.

Figure 10(d) reports the latency of our algorithms for scenario A. As expected, increased depth (conjunctions with increasing number of types) leads to increased latency. Here the "depth" k is fixed to 4, but latency is reported independently at different depths. The observed latency, averaged over all subscribers within each level, was approximately the same with replicated and non-replicated mergers.

6.3 Disjunctions

Figure 10(e) compares the scalability of FAIDECS with respect to throughput in scenario D. The 3 curves represent different depths of the hierarchy (between 2 to 4 levels). For each curve, the throughput is averaged at the respective level. We observe that the impact on throughput is minimal when the disjunctions are made more complex. As shown in Figure 10(f), the latency for 4 types improves slightly. This is because disjunctions provide more than one possibility for event delivery, and the system is no longer throttled by the rate of the slowest upstream process as with conjunctions.

7 Conclusions

We have presented decentralized algorithms for event correlation implemented in FAIDECS. Our algorithms provide clear properties, hinging on a novel notion of subscription subsumption tailored to correlation. The same properties can be achieved by less specialized solutions such as sequencer-based schemes, yet our solutions are inherently more scalable and reliable, leading to strong properties with practical performance; our solutions are also more scalable than peer-based approaches, e.g., relying on tokens, while still achieving practical fault-tolerance. We are currently exploring extensions of our algorithms and additional properties (e.g., causal order).

References

1. D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 2003.

2. M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching Events in a Content-Based Subscription System. *PODC*, 1999.
3. A. Basu, B. Charron-Bost, and S. Toueg. Simulating Reliable Links with Unreliable Links in the Presence of Failures. *WDAG*, 1996.
4. K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM TOCS*, 1999.
5. A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide Area Event Notification Service. *ACM TOCS*, 2001.
6. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. *VLDB*, 1994.
7. X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM CSUR*, 2004.
8. A.J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W.M. White. Towards Expressive Publish/Subscribe Systems. *EDBT*, 2006.
9. P. Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. *ECOOP*, 2009.
10. C. L. Forgy. On the efficient implementation of production systems. PhD thesis, Carnegie Mellon University, 1979.
11. H. Garcia-Molina and A. Spauster. Message Ordering in a Multicast Environment. *ICDCS*, 1989.
12. R. Guerraoui and A. Schiper. Genuine Atomic Multicast in Asynchronous Distributed Systems. *TCS*, 2001
13. R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. E. Anderson, B. N. Bershad, G. Borriello, S. D. Gribble, and D. Wetherall. System Support for Pervasive Applications. *ACM TOCS*, 2004.
14. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. *Distributed Systems*, 2nd edition, 1993.
15. G. G. Koch, B. Koldehofe, and K. Rothermel. Cordies: Expressive Event Correlation in Distributed Systems. *DEBS*, 2010.
16. R. R. Kompella, J. Yates, A. G. Greenberg, and A. C. Snoeren. IP Fault Localization Via Risk Modeling. *NSDI*, 2005.
17. C. Krugel, T. Toth, and C. Kerer. Decentralized Event Correlation for Intrusion Detection. *ICISC*, 2002.
18. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 1978.
19. G. Li and H. Jacobsen. Composite Subscriptions in Content-Based Publish/Subscribe Systems. *Middleware*, 2005.
20. P.R. Pietzuch, B. Shand, and J. Bacon. A Framework for Event Composition in Distributed Systems. *Middleware*, 2003.
21. E. Rabinovich, O. Etzion, S. Ruah, and S. Archushin. Analyzing the Behavior of Event Processing Applications. *DEBS*, 2010.
22. C. Sanchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dill, and Z. Manna. Event Correlation: Language and Semantics. *EMSOFT*, 2003.
23. N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. *VLDB*, 2007.
24. G.A. Wilkin and P. Eugster, Multicast with Aggregated Deliveries. `http://www.cs.purdue.edu/homes/peugster/EventJava/MDMcastTR.pdf`, 2010.
25. G.A. Wilkin, K.R. Jayaram, P. Eugster and A. Khetrapal, Fair Decentralized Event Correlation with FAIDECS. `http://www.cs.purdue.edu/homes/peugster/EventJava/FAIDECSTR.pdf`, 2011.
26. Y. Zhao and R.E. Strom. Exploiting Event Stream Interpretation in Publish-Subscribe Systems. *PODC*, 2001.