Toward Dynamism in Distributed Lingua Franca Programs

Chadlia Jerad and Edward A. Lee

***Abstract*—Distributed systems often require dynamic capabilities to ensure adaptability, efficiency, and fault-tolerance. In applications where determinism and timing are crucial, a clear and well-defined approach to deterministic dynamism is much needed, but inherently difficult to define. This work gives dynamism deterministic semantics, thus enabling precise and repeatable behavior. To this end, we select the Lingua Franca (LF) coordination language that is based on the reactor model, and introduce dynamism to the distributed LF programs, referred to as federations. This paper outlines the challenges associated with incorporating transient federates, which are capable of joining and leaving the federation at arbitrary times, and proposes solutions to the identified problems. A realistic example of an online auction system is used to illustrate the approach. Furthermore, the potential applications of this mechanism are discussed, along with the challenges that need to be addressed.**

***Index Terms*—Reactor Model, Distributed Systems, Determinism, Dynamism.**

I. Introduction

THE actor model has been widely adopted for building distributed systems, thanks to its inherent parallelism and message-driven nature that emphasizes its potential for large-scale distributed systems [1]. This comes, however, at the cost of nondeterminism, which often leads to inconsistency. Moreover, the subtleties in defining *time* across physically distributed communicating processes allow only for partially logically ordered clocks [2]. The recently proposed reactor model by Lohstroh et al. [3] offers a promising alternative that enables deterministic concurrency with time as a first class citizen [4]. Lingua Franca (LF) [5], a coordination language based on the reactor model, facilitates deterministic concurrency and leverages parallelism transparently. Furthermore, Menard et al. show that the runtime performance of LF programs, in terms of execution time, surpasses that of popular actor frameworks such as Akka and CAF [6].

Motivation: Distributed LF programs, known as federations, currently operate as single monolithic applications spread across machines [7]. When a federation starts, all federates that will ever join must do so at the beginning; otherwise, the federation will not start. Additionally, the current design assumes that if any federate leaves, the entire federation must shut down. While this design suits certain applications, many others require more flexibility. Other applications have to tolerate potentially unbounded network delay, possible link failures, and component failures, while others intrinsically serve a number of components that can vary over time. Therefore, support for a certain level of dynamism is needed.

Determinism in Dynamic Systems: It may seem contradictory to talk about determinism for dynamic distributed systems where components can join and leave at unexpected times. Our key contribution is to give this dynamism deterministic semantics, thereby enabling precise and repeatable behavior. To justify this claim, first note that determinism is a property of a model, not a physical realization [8]. The model, in this case, assigns a logical time to the events of joining or leaving a federation, and then our implementation ensures that all participants see these events in logical time order w.r.t. any other logically-timed events. Moreover, we show how these assigned logical times can be derived from imperfectly synchronized physical clocks without compromising determinism, so that the logical times assigned to these events represent reasonable measures of the physical times of those events. Among other benefits, this enables regression testing, where a test system controls the logical time at which components join and leave a federation, and exactly one known-good-behavior emerges.

Paper Outline: After introducing Lingua Franca through the motivational example of an online auction system in Sec. II, we identify and formulate the requirements governing the semantics of the federated execution in LF in Sec. III. We derive the key challenges along with the adopted solutions for supporting the dynamic behavior of transient federates (Sec. IV). We discuss the particular case of timers alignment in Sec. V. We finally discuss the potential applications of the introduced mechanism in Sec. VI.

II. LF through a Motivational Example

This section introduces Lingua Franca through a practical example involving a distributed online auction system (OAS). To make it simple, we assume that all bids are binding and consider only one item to be auctioned in an execution. Furthermore, we define the *silence time* as the time duration the auctioneer waits before making a decision. This duration is reset each time a bidder either joins the auction or places a bid. In order to illustrate the deterministic and repeatable distributed execution, we push the design to the limits. In this test case, all bidders place new bids at the last possible moment, resulting in numerous simultaneous bids. We consider any bid made at the precise moment when the silence time expires to be valid, thereby triggering further bidding activity.

Fig. 1 shows a graphical description of an LF program with one Auctioneer component and any number of Bidder components. This is a federation, as indicated by the cloud icon next to the name on top, meaning that the components

Chadlia Jerad is with the University of Manouba, Manouba, Tunisia (e-mail: chadlia.jerad@ensi-uma.tn).

Edward A. Lee is with the University of California at Berkeley, CA, USA (e-mail: eal@berkeley.edu)

execute as separate processes, possibly on separate machines. Our test case defines five reactor instances: `bid` (an array of size four), and `auc`, that send each other tagged messages (events) via named ports. Reactors define *state* variables that manipulate encapsulated data and *actions* that are used for scheduling events in the future. Timers are one special case of actions that schedule events periodically or one-time. Events are ordered on a logical timeline, and each is assigned a *tag*. Tags are pairs of a time value and a microstep index, thus enabling the use of superdense time.

In the example, `bid[i]`, $i \in \{0, 1, 2, 3\}$ are instances of `Bidder`, which represent users that place bids and send them to the auctioneer. Instances of `Bidder` define three `reactions` (depicted by dark gray chevrons) that contain code that is written in one of the supported target languages, C, C++, Python, Rust or TypeScript. In case of OAS, we use C. Reactions, when triggered by inputs, execute atomically and may produce outputs or schedule actions. In the diagram, the dashed lines show triggers and effects of reactions. `Bidder` uses a logical action to trigger a bid, indicated by the triangle labeled 'L'. Reaction #1 in `Bidder` is triggered on startup and notifies the auctioneer that it joined. It also triggers the logical action with *zero delay*, which triggers reaction #2 one microstep later. Reaction #2 decides whether to place a bid (at random), and if so, sends it through `outBid` port and then reschedules itself with a delay equal to the silence time. Reaction #3 updates the internal state variables with incoming information from the auctioneer about the latest bids or situation. Reaction numbering within each reactor in the diagram reflects the deterministic ordering of their execution, which, in case of events or messages with identical tags, will follow the lexical order of the reaction definitions in the source code. Consequently, when placing a bid at tag t, the bidder will not see the decision at t before placing their bid. The bidder cannot be sure they are making a winning bid because other bidders could simultaneously make a bid.

Reactor `Auctioneer`, manages the auction process. `Auctioneer` defines three reactions. The startup reaction #1 sets the minimum bid (at random). Reaction #2 reacts to received bids. If this is a joining bidder, identified by a bidding amount of 0, then it will be notified with the current status of the auction. Reaction #2 observes all bids received at a tag. If a bid is higher than previous bids, it notifies all bidders and schedules the logical action to observe the silence time. Reaction #3, when triggered, proceeds with a decision.

Three rules govern the intended behavior. If bidding starts and no bidder places bids before the silence timeout, then the item is withdrawn and bidding stops. If bids have been placed, and the silence time passes without a new big, bidding is concluded and the latest maximum is the winner. Otherwise, bidding continues. The decision has lower priority than received bids with the same tag. So corner cases of simultaneity are dealt with cleanly, fairly, and repeatably.

In this example, consistency and time are important. Consequently, a *centralized coordination* scheme favoring consistency over availability [9] is used. Logical time and reaction precedence relations give an unambiguous ordering to the bids from the bidders, as well as to the decision making. Given the distributed nature of the system, network failures may occur. It is also possible to have a bidder that decides to leave the auction before it is concluded. Hence, `Bidder` instances need to be *transient*, in the sense that they may join, leave, and re-join at runtime, without causing the system to stop and while preserving determinism. The auction data must remain consistent in that all observers must agree on the highest bid at any logical time.

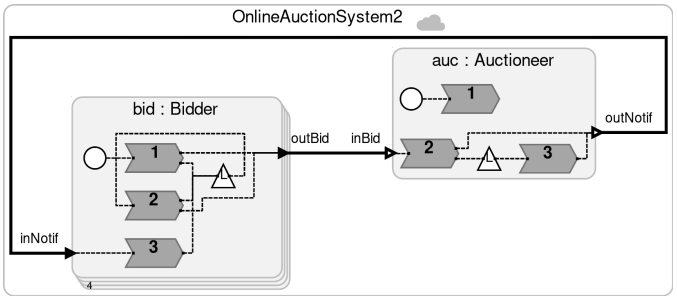


Fig. 1. Online Auction System Example.

III. Requirements

This section introduces the requirements governing the execution of distributed LF programs, which are essential to discuss how dynamism can be supported while preserving determinism (c.f. Sec. IV).

The first requirement Req. #1 establishes the relationship between logical time and physical time. Let Σ be the set of events in the execution of an LF program. Let ϕ, τ, τ_t, and τ_m be the functions that, for an event $e \in \Sigma$, respectively return the physical time, the tag $g = (e_t, e_m)$, the logical time e_t, and the microstep e_m at which the event occurred. By default, the runtime system does its best to keep logical time close to, but lagging behind, physical time. Consequently, logical time is chasing physical time. Formally, $\forall e \in \Sigma, \tau_t(e) \leq \phi(e)$.

Federated execution goes through three phases: *startup*, *execution*, and *shutdown*. The Run Time Infrastructure (RTI) manages the startup and shutdown of the federation. In centralized coordination, all exchanged messages pass through the RTI. In decentralized execution, however, messages are exchanged directly between federates. This work focuses on centralized coordination, deferring the support of transients in decentralized coordination for future research. During the startup phase, when a federate registers with the RTI, it performs clock synchronization and then sends its current physical time. Once the RTI has heard from all expected federates, it selects the maximum of these times, adds an offset, and broadcasts the result. This determines the *logical start time* s of the federation. Federates will wait until their physical clock matches s to transition to the execution phase. As a result, if the offset is sufficiently high, federates will start executing at approximately the same physical time, which is close to the starting logical time. It is possible though that the network delay exceeds the offset. In either case, the logic of the program is unaffected. The shutdown phase is similarly coordinated. Consequently, Req. #2 specifies that the RTI and the federates have synchronized clocks with bounded error.

In centralized coordination, the RTI regulates the advancement of the logical time for each federate. It serves as the *guardian* of LF semantics and the *time arbiter*. Before a federate processes reactions at a given tag, it must first obtain RTI approval. When a federate f completes executing a tag, it notifies the RTI with a *Logical Tag Completed* (LTC) signal. To request permission to process a tag, the federate sends a *Next Event Tag* (NET) signal to the RTI. These signals prompt the RTI to execute a decision algorithm to derive and issue *Tag Advance Grant* (TAG) or *Provisional Tag Advance Grant* (PTAG) signals to f and its downstream federates. If f has no upstream federates, the RTI will always grant time advance, and f will not wait for the RTI response.

In distributed settings, relationships involving time are challenging, making it difficult to know a state across all system components. For this, the requirements governing exchanged messages will be expressed per observer. When the RTI is the observer, at any time instant t, Req. #3 specifies that the TAG of a federate is higher or equal than the TAG of its downstream federates. Formally, $\forall f' \in Downstream(f), TAG(f') \leq TAG(f)$. Req. #4, on the other hand, specifies that the RTI knows about the most recent TAG of f before f itself. When a federate f is the observer, however, at any time instant t, Req. #5 specifies that f knows about its most recent LTC, NET, and current tag before the RTI knows.

IV. Supporting Transient Federates

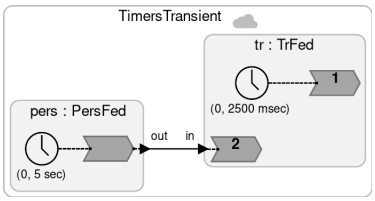
In our extension, federates fall into two types: *persistent* and *transient*. Persistent federates must be present for the federation to start and must last until its shutdown. In other words, their execution lifetime is equal to the federation's execution lifetime. Transient federates, however, can join and leave multiple times during the federation's execution lifetime. They are not required to be present for the federation to start. In the OAS example, `auc` is persistent and `bid[i]`, $i \in \{0, 1, 2, 3\}$ are transient.

In LF, the connection between federates is statically specified, and consequently, a federate that is executing may be connected to a transient federate that is *absent*. But what does it mean for a transient federate to be absent? This means that if a federate sends messages to an absent transient federate, they are dropped by the RTI. To the sending federate, it appears simply that the receiving federate ignores the message. Similarly, an absent federate never sends messages to its downstream federates during the logical time intervals in which it is absent. These time intervals are well defined by the logical time at which the federate left (or the constant `NEVER_TAG` if it has not joined yet) and its *effective joining tag* (see the paragraph following the next one). Once these time intervals are determined, determinism is preserved.

A number of challenges arise in defining the absent and present intervals. We wish to avoid scenarios where the joining of a transient federate results in inconsistent outcomes. For instance, the transient federate sends a tagged message that one executing federate receives, another misses entirely, and a third receives only after advancing to a later tag. The goal is that a transient federate is semantically equivalent to an ordinary LF reactor that happens to ignore inputs in certain tag intervals and not send messages in those same tag intervals.

```
1 target C
2 federated reactor {
3   pers = new PersFed()
4   @transient
5   tr = new TrFed()
6   pers.out -> tr.in
7 }
```

(a)



(b)

Fig. 2. Example Illustrating Timers in Transient Federates.

When a transient federate f wants to join a federation that is running, it should join using the same protocol as for non-transient federates. The RTI will need to choose an *effective start tag* g at which f will start executing. It will have to ensure that no downstream federate has advanced to g or greater (see Req. #3). Downstream federates cannot advance to a particular tag until the RTI sends them a TAG (tag advance grant) or PTAG (provisional TAG) signal, so the RTI can choose the larger of the joining tag $(t_\phi, 0)$, and all (P)TAG signals it has sent to downstream federates. If the maximum tag is equal to any of the (P)TAG signals, one microstep is added to determine an effective start tag g for f.

Moreover, the RTI will have to ensure that no upstream federate has sent a message with tag g or greater in order to grant f the effective start tag g. With centralized coordination, all messages flow through the RTI, so the RTI can choose the maximum tag of messages from upstream federates and add one microstep. The effective start tag g will have to be at least this big. This solution guarantees that Req. #4 holds.

Another challenge is that the RTI currently will send (P)TAG signals to downstream federates with no concern for physical time. A downstream federate may have sent a NET signal with a tag g' that is far in the future compared to physical time, and, if it is safe, the RTI may immediately grant advancement to g'. If there has been such a grant far in the future, then f will be granted an effective start tag that is also far in the future. This will delay f joining the federation.

Our proposal is that for any federate that has an absent upstream transient federate, the RTI delays the granting of any (P)TAG with tag g' until physical time at the RTI has surpassed g' (see Req. #1 and Req. #2). This will minimize the delay of f joining the federation at the expense of a modest increase in lag experienced by downstream federates. This increase in lag is an unavoidable consequence of the CAL theorem [9]. Hence, the RTI maintains a queue of (P)TAG signals to be sent when physical time advances sufficiently. If and when a federate joins, any pending (P)TAG signals waiting to be sent will be canceled and reconsidered when a NET signal is received from the joining transient.

V. Timers Alignment in Transient Federates

Recall from Sec. II that timers in LF are logical actions that schedule events periodically or one-time. The first execution of the timer reaction occurs after a predefined offset. How should timers in a joining transient federate align with timers

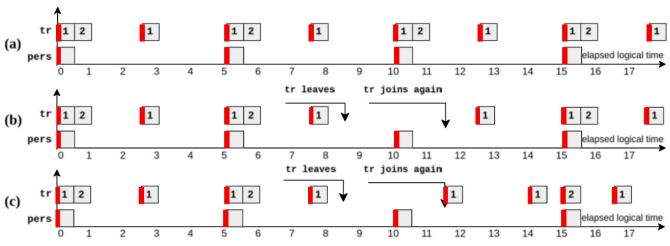


Fig. 3. Proposals for Timers Alignment.

in other federates? Should the offset be executed based of the effective start tag g, or should they start as if they had been running since the federation started at s?

Let's consider the simple example listed graphically and partially in text in Fig. 2. The example defines a federation consisting of two reactors: `pers` and `tr`. `pers` is persistent and defines a reaction triggered by a timer every 5000ms, producing an output at each execution. In contrast, `tr` is transient, with reaction #1 triggered every 2500ms and reaction #2 triggered whenever an input is received at the input port `in`. If `tr` executes as a persistent federate, the execution pattern in logical time will resemble the one shown in Fig. 3-(a), where the red bar signals the discrete jump in logical time. Note that the execution of the reaction of `pers` and reaction #1 of `tr` will be logically simultaneous. Suppose now that `tr` reactor starts with the rest of the federation, but drops out at elapsed logical time 8500ms and then rejoins at elapsed logical time 11500ms, how should the timer reactions align?

There are two choices. Option 1, illustrated in Fig. 3-(b), is to align the timers as if they have been running since the global start time s. Option 2, illustrated in Fig. 3-(c), is to start timers in a transient federate at the effective start time. Option 2 is the adopted one for two reasons. The first is that reactions to the 'startup' event will coincide with the first reactions to any timer with a zero offset, as they do for non-transient federates. Moreover, option 1 can be manually built by the application designer using logical actions.

VI. Limitations, Enhancements, and Further Potential Applications

Hot Swapping: In prototyping transient federates, we realized that they enable a simple *hot swap* mechanism as a small embellishment that follows almost immediately from having solved the problems above. This mechanism is identical to having a federate join late, with the exception that it replaces an existing federate rather than filling an empty slot. In general, this mechanism is an enabler for runtime bug fixes and even feature augmentation. Similar to the transients federates joining at arbitrarily times, these possibilities raise security concerns. Therefore, accepting a transient federate in general, or a hot swap in particular, requires carefully considered security mechanisms. Our implementation works with the pre-existing HMAC authentication available in LF.

Fault Tolerance and State Persistence: Transient federates enable fault tolerance by allowing system recovery after failure, ensuring continued correct operation. Currently, a manually implemented use case has been developed to showcase state persistence across executions. For a complete solution, the LF runtime will need to be augmented with serialization and automatic launch after failure detection.

Transients vs. Mutations: Although the reactor model defines mutations, which are reactions that modify the program structure at runtime, LF does not yet implement them. Our proposal is more modest in that it preserves the (statically analyzable) structure of the distributed program throughout its lifetime and only allows defined nodes to come and go.

Regression Testing: The model of dynamic federations that we provide is deterministic in the sense that once the RTI assigns a tag to a joining event, the reaction of the rest of the system is unique and well defined. However, to perform regression testing, we need to control the tag that the RTI assigns. By default, the RTI uses its local physical clock to assign a logical time. We propose that, for the purposes of regression testing, the RTI is augmented to support federates of type "transient test" that will declare during system startup their times of joining, and the RTI will respect this.

VII. Conclusion and Future Work

This paper presents deterministic semantics for the dynamic execution of distributed programs, enabling precise and repeatable behavior. This is achieved through the Lingua Franca coordination language and its deterministic, time-sensitive reactor model. The model assigns logical times to events of joining or leaving a federation, ensuring that all participants observe these events in logical time order. The given online auction example stresses the semantics by explicitly testing difficult cases of simultaneous events. Our implemented approach promises enhancements, including hot swapping, fault tolerance, and regression testing. In addition, scalability, an important issue, will be explored in future work.

References

[1] B. Hedden and X. Zhao, "A comprehensive study on bugs in actor systems," in *ICPP'18: Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018.

[2] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[3] M. Lohstroh, Í. Íncer Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," in *Cyber Physical Systems. Model-Based Design: 9th International Workshop, CyPhy 2019, and 15th International Workshop, WESE 2019*, 2019, p. 27.

[4] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee, "Invited: Actors revisited for time-critical systems," in *Design Automation Conference (DAC)*, June 2019.

[5] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 4, may 2021.

[6] C. Menard, M. Lohstroh, S. Bateni, M. Chorlian, A. Deng, P. Donovan, C. Fournier, S. Lin, F. Suchert, T. Tanneberger *et al.*, "High-performance deterministic concurrency using lingua franca," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 4, pp. 1–29, 2023.

[7] S. Bateni, M. Lohstroh, H. S. Wong, H. Kim, S. Lin, C. Menard, and E. A. Lee, "Risk and mitigation of nondeterminism in distributed cyber-physical systems," in *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2023.

[8] E. A. Lee, "Determinism," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, pp. 1–34, 2021.

[9] E. A. Lee, R. Akella, S. Bateni, S. Lin, M. Lohstroh, and C. Menard, "Consistency vs. availability in distributed cyber-physical systems," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, sep 2023.