The Future of a Missed Deadline

Behrooz Nobakht, Frank Boer, Mohammad Jaghoori

The Future of a Missed Deadline

Behrooz Nobakht[1,3], Frank S. de Boer[2], and Mohammad Mahdi Jaghoori[1]

[1] Leiden University
bnobakht@liacs.nl, m.jaghouri@lacdr.leidenuniv.nl
[2] Centrum Wiskunde en Informatica
frb@cwi.nl
[3] SDL Fredhopper
bnobakht@sdl.com

Abstract. In this paper, we introduce a real-time actor-based programming language and provide a formal but intuitive operational semantics for it. The language supports a general mechanism for handling exceptions raised by missed deadlines and the specification of application-level scheduling policies. We discuss the implementation of the language and illustrate the use of its constructs with an industrial case study from distributed e-commerce and marketing domain.

Keywords: actors, application-level scheduling, real-time, deadlines, futures, Java.

1 Introduction

In real-time applications, rigid deadlines necessitate stringent scheduling strategies. Therefore, the developer must ideally be able to program the scheduling of different tasks inside the application. Real-Time Specification for Java (RTSJ) [11,12] is a major extension of Java, as a mainstream programming language, aiming at enabling real-time application development. Although RTSJ extensively enriches Java with a framework for the specification of real-time applications, it yet remains at the level of conventional *multithreading*. The drawback of multithreading is that it involves the programmer with OS-related concepts like threads, whereas a real-time Java developer should only be concerned about high-level entities, i.e., objects and method invocations, also with respect to real-time requirements.

The actor model [9] and actor-based programming languages, which have re-emerged in the past few years [24,3,10,14,26], provide a different and promising paradigm for concurrency and distributed computing, in which threads are transparently encapsulated inside actors. As we will argue in this paper, this paradigm is much more suitable for real-time programming because it enables the programmer to obtain the appropriate high-level view which allows the management of complex real-time requirements.

In this paper, we introduce an actor-based programming language Crisp for real-time applications. Basic real-time requirements include deadlines and time-

outs. In Crisp, deadlines are associated with asynchronous messages and timeouts with futures [6]. Crisp further supports a general actor-based mechanism for handling exceptions raised by missed deadlines. By the integration of these basic real-time control mechanisms with the application-level policies supported by Crisp for scheduling of the messages inside an actor, more complex real-time requirements of the application can be met with more flexibility and finer granularity.

We formalize the design of Crisp by means of structural operational semantics [22] and describe its implementation as a full-fledged programming language. This implementation uses both the Java and Scala language with extensions of Akka library. We illustrate the use of the programming language with an industrial case study from SDL Fredhopper that provides enterprise-scale distributed e-commerce solutions on the cloud.

The paper continues as follows: Section 2 introduces the language constructs and provides informal semantics of the language with a case study in Section 2.1. Section 3 presents the operational semantics of Crisp. Section 4 follows to provide a detailed discussion on the implementation. The case study continues in this section with further details and code examples. Section 5 discusses related work of research and finally Section 6 concludes the paper and proposes future line of research.

2 Programming with deadlines

In this section, we introduce the basic concepts underlying the notion of "deadlines" for asynchronous messages between actors. The main new constructs specify how a message can be sent with a deadline, how the message response can be processed, and what happens when a deadline is missed. We discuss the informal semantics of these concepts and illustrate them using a case study in Section 2.1.

Listing 1 introduces a minimal version of the real-time actor-based language Crisp. Below we discuss the two main new language constructs presented at lines (7) and (8).

How to send a message with a deadline? The construct

$$f = e_0 \,!\, m(\overline{e})\ \texttt{deadline}(e_1)$$

describes an asynchronous message with a deadline specified by e_1 (of type T_{time}). Deadlines can be specified using a notion of time unit such as millisecond, second, minute or other units of time. The caller expects the callee (denoted by e_0) to process the message within the units of time specified by e_1. Here processing a message means starting the execution of the process generated by the message. A deadline is missed if and only if the callee does not start processing the message within the specified units of time.

What happens when a deadline is missed? Messages received by an actor

$$C ::= \textbf{class}\ \mathsf{N}\ \textbf{begin}\ V^{?}\ \{M\}^{*}\ \textbf{end} \quad (1)$$
$$M_{sig} ::= \mathsf{N}(\overline{T\ x}) \quad (2)$$
$$M ::= \{M_{sig} == \{V\ ;\}^{?} S\} \quad (3)$$
$$V ::= \textbf{var}\ \{\{x\},^{+}\ :\ \mathsf{T}\ \{= e\}^{?}\},^{+} \quad (4)$$
$$S ::= x := e\ | \quad (5)$$
$$::= x := \textbf{new}\ \mathsf{T}(e^{?})\ | \quad (6)$$
$$:= f = e\ !\ m(\overline{e})\ \textbf{deadline}(e)\ | \quad (7)$$
$$::= x := f.\textbf{get}(e^{?})\ | \quad (8)$$
$$::= \textbf{return}\ e\ | \quad (9)$$
$$::= S\ ;\ S\ | \quad (10)$$
$$::= \textbf{if}\ (b)\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end}\ | \quad (11)$$
$$::= \textbf{while}\ (b)\ \{\ S\ \}\ | \quad (12)$$
$$::= \textbf{try}\ \{S\}\ \textbf{catch}(\mathsf{T}_{\texttt{Exception}}\ x)\ \{\ S\ \} \quad (13)$$

Fig. 1: A kernel version of the real-time programming language. The bold scripted **keywords** denote the reserved words in the language. The over-lined $\overline{v}$ denotes a sequence of syntactic entities v. Both local and instance variables are denoted by x. We assume distinguished local variables `this`, `myfuture`, and `deadline` which denote the actor itself, the unique future corresponding to the process, and its deadline, respectively. A distinguished instance variable `time` denotes the current time. Any subscripted type $\mathsf{T}_{specialized}$ denotes a *specialized* type of general type T; e.g. $\mathsf{T}_{\texttt{Exception}}$ denotes all "exception" types. A variable f is in T_{future}. N is a name (identifier) used for classes and method names. C denotes a class definition which consists of a definition of its instance variables and its methods; M_{sig} is a method signature; M is a method definition; S denotes a statement. We abstract from the syntax the side-effect free expressions e and boolean expressions b.

generate processes. Each actor contains one active process and all its other processes are queued. Newly generated processes are *inserted* in the queue according to an application-specific policy. When a *queued* process misses its deadline it is *removed* from the queue and a corresponding *exception* is recorded by its future (as described below). When the currently active process is terminated the process at the head of the queue is activated (and as such dequeued). The active process cannot be *preempted* and is forced to *run to completion*. In Section 4 we discuss the implementation details of this design choice.

How to process the response of a message with a deadline? In the above example of an asynchronous message, the *future* result of processing the message is denoted by the variable f which has the type of **`Future`**. Given a future variable f, the programmer can query the availability of the result by the construct

$$v = f.\texttt{get}(e)$$

The execution of the **get** operation terminates successfully when the future variable f contains the result value. In case the future variable f records an exception, e.g. in case the corresponding process has missed its deadline, the **get** operation is *aborted* and the exception is *propagated*. Exceptions can be caught by `try-catch` blocks.

Listing 1: Using try-catch for processing future values

```
1 try {
2   x = f.get(e)
3   S_1
4 } catch(Exception x) {
5   S_2
6 }
```

For example, in Listing 1, if the **get** operation raises an exception control, is transferred to line (5); otherwise, the execution continues in line (3). In the `catch` block, the programmer has also access to the occurred exception that can be any kind of exception including an exception that is caused by a missed deadline. In general, any uncaught exception gives rise to abortion of the active process and is recorded by its future. Exceptions in our actor-based model thus are propagated by futures.

The additional parameter e of the get operation is of type T_{time} and specifies a *timeout*; i.e., the **get** operation will timeout after the specified units of time.

2.1 Case Study: Fredhopper Distributed Data Processing

Fredhopper is an SDL company since 2008 and a leading search, merchandising and personalization solution provider, whose products are uniquely tailored to the needs of online business. Fredhopper operates behind the scenes of more than 100 of the largest online sellers. The Fredhopper Access Server (FAS) provides access to high quality product catalogs. Typically deployments have about 10 explicit attribute values associated with a product over thousands of attribute dimensions. This challenging task involves working on difficult issues, such as the performance of information retrieval algorithms, the scalability of dealing with huge amounts of data and in satisfying large amounts of user requests per unit of time, the fault tolerance of complex distributed systems, and the executive monitoring and management of large-scale information retrieval oper-

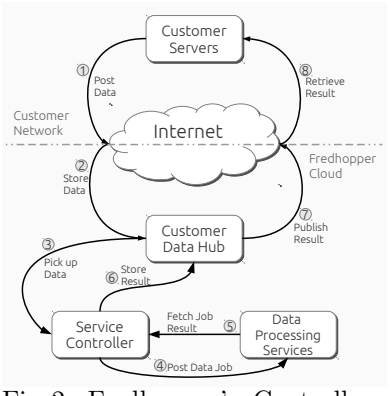

Fig. 2: Fredhopper's Controller life cycle for remote data processing

ations. Fredhopper offers its services and facilities to e-Commerce companies (customers) as services (SaaS) over the cloud computing infrastructure (IaaS); which gives rise to different challenges in regards with resources management techniques and the customer cost model and service level agreements (SLA).

To orchestrate different services such as FAS or data processing, Fredhopper takes advantage of a service controller (a.k.a. Controller). Controller is responsible to passively manage different service installations for each customer. For instance, in one scenario, a customer submits their data along with a processing request to their data hub server. Controller, then picks up the data and initiates a data processing job (usually an ETL job) in a data processing service. When the data processing is complete, the result is again published to customer environment and additionally becomes available through FAS services. Figure 2 illustrates an example scenario that is described above.

In the current implementation of Controller, at Step 4, a data job instance is submitted to a remote data processing service. Afterwards, the future response of the data job is determined by a periodic remote check on the data service (Step 4). When the job is finished, Controller continues to retrieve the data job results (Step 5) and eventually publishes it to customer environment (Step 6).

In terms of system responsiveness, Step 4 may never complete. Step 4 failure can have different causes. For instance, at any moment of time, there are different customers' data jobs running on one data service node; i.e. there is a chance that a data service becomes overloaded with data jobs preventing the periodic data job check to return. If Step 4 fails, it leads the customer into an *unbounded waiting* situation. According to SLA agreements, this is *not* acceptable. It is strongly required that for any data job, the customer should be notified of the result: either a completed job with *success/failed* status, a job that is not completed, or a job with an unknown state. In other words, Controller should be able to guarantee that any data job request terminates.

To illustrate the contribution of this paper, we extract a closed-world simplified version of the scenario in Figure 2 from Controller. In Section 4, we provide an implementation-level usage of our work applied to this case study.

3 Operational Semantics

We describe the semantics of the language by means of a two-tiered labeled transition system: a local transition system describes the behavior of a single actor and a global transition system describes the overall behavior of a system of interacting actors. We define an actor state as a pair $\langle p, q \rangle$, where

- p denotes the current active process of the actor, and
- q denotes a queue of pending processes.

Each pending process is a pair (S, τ) consisting of the current executing statement S and the assignment τ of values to the *local* variables (e.g., formal parameters). The active process consists of a pair (S, σ), where σ assigns values

to the local variables and additionally assigns values to the instance variables of the actor.

3.1 Local transition system

The local transition system defines transitions among actor configurations of the form $\langle p, q, \phi \rangle$, where (p, q) is an actor state and for any object o identifying a created future, ϕ denotes the shared heap of the created future objects, i.e., $\phi(o)$, for any future object o existing in ϕ, denotes a record with a field val which represents the return value and a boolean field $\mathsf{aborted}$ which indicates abortion of the process identified by o.

In the local transition system we make use of the following axiomatization of the occurrence of exceptions. Here $(S, \sigma, \phi) \uparrow v$ indicates that S raises an exception v:

- $(x = f.\texttt{get}(), \sigma, \phi) \uparrow \sigma(f)$ where $\phi(\sigma(f)).\mathsf{aborted} = \texttt{true}$,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ u)\{S'\} \uparrow v}$ where v is not of type T, and,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{(S; S, \sigma, \phi)' \uparrow v}$.

We present here the following transitions describing internal computation steps (we denote by $\mathsf{val}(e)(\sigma)$ the value of the expression e in σ and by $f[u \mapsto v]$ the result of assigning the value v to u in the function f).

Assignment statement is used to assign a value to a variable:

$$\langle (x = e; S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma[x \mapsto \mathsf{val}(e)(\sigma)]), q, \phi \rangle$$

Returning a result consists of setting the field val of the future of the process:

$$\langle (\texttt{return}\ e; S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma), q, \phi[\sigma(\texttt{myfuture}).\mathsf{val} \mapsto \mathsf{val}(e)(\sigma)] \rangle$$

Initialization of `timeout` in `get` operation assigns to a distinguished (local) variable `timeout` its initial *absolute* value:

$$\langle (x = f.\texttt{get}(e); S, \sigma), q, \phi \rangle \rightarrow \langle (x = f.\texttt{get}(e); S, \sigma[\texttt{timeout} \mapsto \mathsf{val}(e + \mathsf{time})(\sigma), q, \phi \rangle$$

The `get` operation is used to assign the value of a future to a variable:

$$\langle (x = f.\texttt{get}(); S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma[x \mapsto \phi(\sigma(f)).\mathsf{val}]), q, \phi \rangle$$

where $\phi(\sigma(f)).\mathsf{val} \neq \bot$.

Timeout is operationally presented by the following transition:

$$\langle (x = f.\texttt{get}(); S, \sigma), q, \phi \rangle \to \langle (S, \sigma), q, \phi \rangle$$

where $\sigma(\texttt{time}) < \sigma(\texttt{timeout})$.

The `try-catch` block semantics is presented by:

$$\frac{\langle (S, \sigma), q, \phi \rangle \to \langle (S', \sigma'), q', \phi' \rangle}{\langle (\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ x)\{S''\}; S''', \sigma), q, \phi \rangle \to \langle (\texttt{try}\{S'\}\texttt{catch}(\texttt{T}\ x)\{S''\}; S''', \sigma), q', \phi' \rangle}$$

Exception handling. We provide the operational semantics of exception handling in a general way in the following:

$$\frac{(S, \sigma, \phi) \uparrow v}{\langle (\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ x)\{S''\}; S''', \sigma), q, \phi \rangle \to \langle (S''; S''', \sigma[x \mapsto v]), q, \phi \rangle}$$

where the exception v is of type $\texttt{T}$.

Abnormal termination of the active process is generated by an uncaught exception:

$$\frac{(S, \sigma, \phi) \uparrow v}{\langle (S; S', \sigma), q, \phi \rangle \to \langle (S'', \sigma'), q', \phi' \rangle}$$

where $q = (S'', \tau) \cdot q'$ and σ' is obtained from restoring the values of the local variables as specified by τ (formally, $\sigma'(x) = \sigma(x)$, for every instance variable x, and $\sigma'(x) = \tau(x)$, for every local variable x), and $\phi'(\sigma(\texttt{myfuture})).\texttt{aborted} = \texttt{true}$ ($\phi'(o) = \phi(o)$, for every $o \neq \sigma(\texttt{myfuture})$).

Normal termination is presented by:

$$\langle (E, \sigma), q, \phi \rangle \to \langle (S, \sigma'), q', \phi \rangle$$

where $q = (S, \tau) \cdot q'$ and σ' is obtained from restoring the values of the local variables as specified by τ (see above). We denote by E termination (identifying $S; E$ with S).

Deadline missed. Let (S', τ) be some pending process in q such that $\tau(\texttt{deadline}) < \sigma(\texttt{time})$. Then

$$\langle (S, \sigma), q, \phi \rangle \to \langle p, q', \phi' \rangle$$

where q' results from q by removing (S', τ) and $\phi'(\tau(\texttt{myfuture})).\texttt{aborted} = \texttt{true}$ ($\phi'(o) = \phi(o)$, for every $o \neq \tau(\texttt{myfuture})$).

A message $m(\tau)$ specifies for the method m the initial assignment τ of its local variables (i.e., the formal parameters and the variables `this`, `myfuture`, and `deadline`). To model locally incoming and outgoing messages we introduce the following labeled transitions.

Incoming message. Let the active process p belong to the actor $\tau(\texttt{this})$ (i.e., $\sigma(\texttt{this}) = \tau(\texttt{this})$ for the assignment σ in p):

$$\langle p, q, \phi\rangle \xrightarrow{m(\tau)} \langle p, \mathsf{insert}(q, m(\overline{v}, d)), \phi\rangle$$

where $\mathsf{insert}(q, m(\tau))$ defines the result of inserting the process (S, τ), where S denotes the body of method m, in q, according to some application-specific policy (described below in Section 4).

Outgoing message. We model an outgoing message by:

$$\langle (f = e_0 \,!\, m(\bar{e})\ \mathbf{deadline}(e_1); S, \sigma), q, \phi\rangle \xrightarrow{m(\tau)} \langle (S, \sigma[f \mapsto o]), q, \phi'\rangle$$

where

- ϕ' results from ϕ by extending its domain with a new future object o such that $\phi'(o).\mathsf{val} = \bot$[4] and $\phi'(o).\mathsf{aborted} = \texttt{false}$,
- $\tau(\texttt{this}) = \mathsf{val}(e_0)(\sigma)$,
- $\tau(x) = \mathsf{val}(e)(\sigma)$, for every formal parameter x and corresponding actual parameter e,
- $\tau(\texttt{deadline}) = \sigma(\mathsf{time}) + \mathsf{val}(e_1)(\sigma)$,
- $\tau(\texttt{myfuture}) = o$.

3.2 Global transition system

A (global) system configuration S is a pair (Σ, ϕ) consisting of a set Σ of actor states and a global heap ϕ which stores the created future objects. We denote actor states by s, s', s'', etc.

Local computation step. The interleaving of local computation steps of the individual actors is modeled by the rule:

$$\frac{(s, \phi) \to (s', \phi')}{(\{s\} \cup \Sigma, \phi) \to (\{s'\} \cup \Sigma, \phi')}$$

Communication. Matching a message sent by one actor with its reception by the specified callee is described by the rule:

$$\frac{(s_1, \phi) \xrightarrow{m(\tau)} (s_1', \phi') \quad (s_2, \phi) \xrightarrow{m(\tau)} (s_2', \phi)}{(\{s_1, s_2\} \cup \Sigma, \phi) \to (\{s_1', s_2'\} \cup \Sigma, \phi')}$$

[4] $\bot$ stands for "uninitialized"

Note that only an outgoing message affects the shared heap ϕ of futures.

Progress of Time. The following transition uniformly updates the local clocks (represented by the instance variable `time`) of the actors.

$$(\Sigma, \phi) \rightarrow (\Sigma', \phi)$$

where

$$\Sigma' = \{\langle (S, \sigma'), q, \phi \rangle \mid \langle (S, \sigma), q, \phi \rangle \in \Sigma,\ \sigma' = \sigma[\texttt{time} \mapsto \sigma(\texttt{time}) + \delta]\}$$

for some positive δ.

4 Implementation

We base our implementation on Java's concurrent package: `java.util.concurrent`. The implementation consists of the following major components:

1. An extensible language API that owns the core abstractions, architecture, and implementation. For instance, the programmer may extend the concept of a scheduler to take full control of how, i.e., in what order, the processes of the individual actors are queued (and as such scheduled for execution). We illustrate the scheduler extensibility with an example in the case study below.
2. Language Compiler that translates the modeling-level programs into Java source. We use ANTLR [21] parser generator framework to compile modeling-level programs to actual implementation-level source code of Java.
3. The language is seamlessly integrated with Java. At the time of programming, language abstractions such as data types and third-party libraries from either Crisp or Java are equally usable by the programmer.

We next discuss the underlying deployment of actors and the implementation of real-time processes with deadlines.

Deploying actors onto JVM threads. In the implementation, each actor owns a main thread of execution, that is, the implementation does *not* allocate *one* thread per process because threads are *costly* resources and allocating to each process one thread in general leads to a poor performance: there can be an arbitrary number of actors in the application and each may receive numerous messages which thus give rise to a number of threads that goes beyond the limits of memory and resources. Additionally, when processes go into pending mode, their correspondent thread may be reused for other processes. Thus, for better performance and optimization of resource utilization, the implementation assigns a single thread for all processes inside each actor.

Consequently, at any moment in time, there is only one process that is executed inside each actor. On the other hand, the actors share a thread which is

used for the execution of a watchdog for the deadlines of the queued processes (described below) because allocation of such a thread to each actor in general slows down the performance. Further this sharing allows the implementation to decide, based on the underlying resources and hardware, to optimize the allocation of the watchdog thread to actors. For instance, as long as the resources on the underlying hardware are abundant, the implementation decides to share as less as possible the watchdog thread. This gives each actor a better opportunity with higher precision to detect missed deadlines.

Implementation of processes with deadlines. A process itself is represented in the implementation by a data structure which encapsulates the values of its local variables and the method to be executed. Given a relative deadline d as specified by a call we compute at run-time its absolute deadline (i.e. the expected starting time of the process) by

$$\texttt{TimeUnit.toMillis}(d) + \texttt{System.currentTimeMillis()}$$

which is a *soft* real-time requirement. As in the operational semantics, in the real-time implementation always the head of the process queue is scheduled for execution. This allows the implementation of a *default* earliest deadline first (EDF) scheduling policy by maintaining a queue ordered by the above absolute time values for the deadlines.

The important consequence of our non-preemptive mode of execution for the implementation is the resulting simplicity of thread management because preemption requires additional thread interrupts that facilitates the abortion of a process in the middle of execution. As stated above, a single thread in the implementation detects if a process has missed its deadline. This task runs periodically and to the end of all actors' life span. To check for a missed deadline it suffices to simply check for a process that the above absolute time value of its deadline is *smaller* than `System.currentTimeMillis()`. When a process misses its deadline, the actions as specified by the corresponding transition of the operational semantics are subsequently performed. The language API provides extension points which allow for each actor the definition of a customized watchdog process and scheduling policy (i.e., policy for enqueuing processes). The customized watchdog processes are still executed by a single thread.

Fredhopper case study. As introduced in Section 2.1, we extract a closed-world simplified version from Fredhopper Controller. We apply the approach discussed in this paper to use deadlines for asynchronous messages.

Listing 2 and 3 present the difference in the previous Controller and the approach in Crisp. The left code snippet shows the Controller that uses polling to retrieve data processing results. The right code snippet shows the one that uses messages with deadlines.

Listing 2: With polling

```
1  class DataProcessor begin
2    op process(d: Data) ==
3      var p := allocDataProcessor(d)
4      p ! process (d)
5      do {
6        s := p ! getStatus (d)
7        if (s <> nil)
8          var r := p ! getResults(d)
9          publishResult(r)
10       wait(TimeUnit.toSecond(1))
11     } while (true)
12 end
```

Listing 3: With deadlines

```
1  class DataProcessor begin
2    op process(d: Data) ==
3      var p := allocDataProcessor(d)
4      var D := estimateDeadline(d)
5      var f :=
6        p ! process (d) deadline (D)
7      try {
8        publishResult(f.get())
9      } catch (Exception x) {
10       if (f.isAborted)
11         notifyFailure(d)
12     }
13 end
```

When the approach in Crisp in the right snippet is applied to Controller, it is guaranteed that all data job requests are terminated in a *finite* amount of time. Therefore, there cannot be complains about never receiving a response for a specific data job request. Many of Fredhopper's customers rely on data jobs to eventually deliver an e-commerce service to their end users. Thus, to provide a guarantee to them that their job result is always published to their environment is critical to them. As shown in the code snippet, if the data job request is failed or aborted based on a deadline miss, the customer is still eventually informed about the situation and may further decide about it. However, in the previous version, the customer may never be able to react to a data job request because its results are never published.

In comparison to the Controller using polling, there is a way to express timeouts for future values. However, it does not provide language constructs to specify a deadline for a message that is sent to data processing service. A deadline may be simulated using a combination of timeout and periodic polling approaches (Listing 2). Though, this approach cannot guarantee eventual termination in all cases; as discussed before that Step 4 in Figure 2 may never complete. Controller is required to meet certain customer expectations based on an SLA. Thus, Controller needs to take advantage of a language/library solution that can provide a higher level of abstraction for real-time scheduling of concurrent messages. When messages in Crisp carry a deadline specification, Controller is able to guarantee that it can provide a response to the customer. This termination guarantee is crucial to the business of the customer.

Additionally, on the data processing service node, the new implementation takes advantage of the extensibility of schedulers in Crisp. As discussed above, the default scheduling policy used for each actor is EDF based on the deadlines carried by incoming messages to the actor. However, this behavior may be extended and replaced by a custom implementation from the programmer. In this case study, the priority of processes may differ if they the job request comes from specific customer; i.e. apart from deadlines, some customers have priority over others because they require a more real-time action on their job requests while others run a more relaxed business model. To model and implement this custom behavior, a custom scheduler is developed for the data processing node.

Listing 4: Data Processor class

```
1 class DataProcessor begin
2   var scheduler := new
        DataScheduler()
3   op process(d: Data) ==
4     // do process
5 end
```

Listing 5: Custom scheduler

```
1 class DataScheduler extends
      DefaultSchedulingManager {
2   boolean isPrior(Process p1,
        Process p2) {
3     if (p1.getCustomer().equals("A
          ")) {
4       return true;
5     }
6     return super.isPrior(p1, p2);
7   }
8 }
```

In the above listings, Listing 5 defines a custom scheduler that determines the priority of two processes with custom logic for specific customer. To use the custom scheduler, the only requirement is that the class `DataProcessor` defines a specific class variable called `scheduler` in Listing 4. The *custom scheduler* is picked up by Crisp core architecture and is used to schedule the queued processes. Thus, all processes from customer `A` have priority over processes from other customers no matter what their deadlines are.

We use Controller's logs for the period of February and March 2013 to examine the evaluation of Crisp approach. We define *customer satisfaction* as a property that represents the effectiveness of futures with deadline. For a customer c, the satisfaction can be denoted by $s = \frac{r_c^F}{r_c}$; in which r_c^F is the number of finished data processing jobs and r_c is the total number of requested data processing jobs from customer c. We extracted statistics for completed and never-ended data processing jobs from Controller logs (s_1). We replayed the logs with Crisp approach and measured the same property (s_2). We measured the same property for 180 customers that Fredhopper manages on the cloud. In this evaluation, a total number of about 25000 data processing requests were included. The results show 6% improvement in Table 1 (that amounts to around 1600 better data processing requests). Because of data issues or wrong parameters in the data processing requests, there are requests that still fail or never end and should be handled by a human resource.

s_1	s_2
88.71%	94.57%

Table 1: Evaluation Results

You may find more information including documentation and source code of Crisp at `http://nobeh.github.com/crisp`.

5 Related Work

The programming language presented in this paper is a real-time extension of the language introduced in [20]. This new extension features

- integration of asynchronous messages with deadlines and futures with timeouts;
- a general mechanism for handling exceptions raised by missed deadlines;
- high-level specification of application-level scheduling policies; and
- a formal operational semantics.

To the best of our knowledge the resulting language is the first implemented real-time actor-based programming language which formally integrates the above features.

In several works, e.g, [2] and [19], asynchronous messages in actor-based languages are extended with deadlines. However these languages do not feature futures with timeouts, a general mechanism for handling exceptions raised by missed deadlines or support the specification of application-level scheduling policies. Futures and fault handling are considered in the ABS language [13]. This work describes recovery mechanisms for failed get operations on a future. However, the language does not support the specification of real-time requirements, i.e., no deadlines for asynchronous messages are considered and no timeouts on futures. Further, when a get operation on a future fails, [13] does not provide any context or information about the exception or the cause for the failure. Alternatively, [13] describes a way to "compensate" for a failed get operation on future. In [4], a real-time extension of ABS with scheduling policies to model distributed systems is introduced. In contrast to Crisp, Real-Time ABS is an executable *modeling* language which supports the explicit specification of the progress of time by means of duration statements for the *analysis* of real-time requirements. The language does not support however asynchronous messages with deadlines and futures with timeouts.

Two successful examples of actor-based programming languages are Scala and Erlang. Scala [10,1] is a hybrid object-oriented and functional programming language inspired by Java. Through the event-based model, Scala also provides the notion of continuations. Scala further provides mechanisms for scheduling of tasks similar to those provided by concurrent Java: it does not provide a direct and customizable platform to manage and schedule messages received by an individual actor. Additionally, Akka [25] extends Scala's actor programming model and as such provides a direct integration with both Java and Scala. Erlang [3] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [5]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [17] (instead of one global scheduler for the entire application) but it does not, for example, support application-level scheduling policies. In general, none these languages provide a formally defined real-time extension which integrates the above features.

There are well-known efforts in Java to bring in the functionality of asynchronous message passing onto multicore including Killim [24], Jetlang [23], ActorFoundry [15], and SALSA [26]. In [15], the authors present a comparative analysis of actor-based frameworks for JVM platform. Most of these frameworks support futures with timeouts but do not provide asynchronous messages with deadlines, or a general mechanism for handling exceptions raised by missed deadlines. Further, pertaining to the domain of priority scheduling of asynchronous messages, these efforts in general provide a predetermined approach or a limited control over message priority scheduling. As another example, in [18] the use of Java Fork/Join is described to optimize mulicore applications. This work is

also based on a *fixed priority* model. Additionally, from embedded hardware-software research domain, Ptolemy [7,16] is an actor-oriented open architecture and platform that is used to design, model and simulate embedded software. Their approach is hardware software co-design. It provides a platform framework along with a set of tools.

In general, existing high-level programming languages provide the programmer with little real-time control over scheduling. The state of the art allows specifying priorities for threads or processes that are used by the operating system, e.g., Real-Time Specification for Java (RTSJ [11,12]) and Erlang. Specifically in RTSJ, [27] extensively introduces and discusses a framework for application-level scheduling in RTSJ. It presents a flexible framework to allow scheduling policies to be used in RTSJ. However, [27] addresses the problem mainly in the context of the standard multithreading approach to concurrency which in general does not provide the most suitable approach to distributed applications. In contrast, in this paper we have shown that an actor-based programming language provides a suitable formal basis for a fully integrated real-time control in distributed applications.

6 Conclusion and future work

In this paper, we presented both a formal semantics and an implementation of a real-time actor-based programming language. We presented how asynchronous messages with deadline can be used to control application-level scheduling with higher abstractions. We illustrated the language usage with a real-world case study from SDL Fredhopper along the discussion for the implementation. Currently we are investigating further optimization of the implementation of Crisp and the formal verification of real-time properties of Crisp applications using schedulability analysis [8].

References

1. *Coordination Models and Languages*, volume 4467, chapter Actors That Unify Threads and Events, pages 171–190. Springer Berlin / Heidelberg, 2007.
2. L. Aceto, M. Cimini, A. Ingólfsdóttir, A. H. Reynisson, S. H. Sigurdarson, and M. Sirjani. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In *FOCLASA*, pages 1–19, 2011.
3. Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
4. Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 2012.
5. Fábio Corrêa. Actors in a new "highly parallel" world. In *Proc. Warm Up Workshop for ACM/IEEE ICSE 2010*, WUP '09, pages 21–24. ACM, 2009.
6. Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

7. J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144.
8. Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis using two clocks. volume 2619 of *Lecture Notes in Computer Science*, pages 224–239. Springer Berlin Heidelberg, 2003.
9. Scott F. Smith Gul A. Agha, Ian A. Mason and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
10. Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
11. JCP. *RTSJ v1 JSR 1*, 1998. `http://jcp.org/en/jsr/detail?id=1`.
12. JCP. *RTSJ v1.1 JSR 282*, 2005. `http://jcp.org/en/jsr/detail?id=282`.
13. Einar Broch Johnsen, Reiner Hhnle, Jan Schfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer Berlin Heidelberg, 2012.
14. Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):39–58, 2007.
15. Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proc. 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.
16. Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems and Computers*, 12(03):231–260, 2003.
17. K Lundin. Inside the Erlang VM, focusing on SMP. Presented at Erlang User Conference. Available at `http://www.erlang.se/euc/08/euc_smp.pdf`, November 13, 2008.
18. Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Combining rtsj with fork/join: a priority-based model. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 82–86, New York, NY, USA, 2011. ACM.
19. Brian Nielsen and Gul Agha. Semantics for an Actor-Based Real-Time Language. In *In Fourth International Workshop on Parallel and Distributed Real-Time Systems*, 1996.
20. Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. 2012. ACM SAC.
21. Terence Parr. Antlr. `http://antlr.org/`.
22. Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61(0):3–15, 2004.
23. Mike Rettig. *Jetlang Library*, 2008. `http://code.google.com/p/jetlang/`.
24. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP 2008 - Object-Oriented Programming*, volume 5142, pages 104–128. Springer Berlin / Heidelberg, 2008.
25. TypeSafe. *Akka*, 2010. `http://akka.io/`.
26. Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36:20–34, December 2001.
27. Alexandros Zerzelidis and Andy Wellings. A framework for flexible scheduling in the RTSJ. *ACM Trans. Embed. Comput. Syst.*, 10(1):3:1–3:44, 2010.