This copy of the Lingua Franca handbook for the ts target was created on Sunday, May 21, 2023 against commit [e7d13d](e7d13d).

Table of Contents

A First Reactor

See the [requirements](#) for using this target.

Minimal Example

A minimal but complete Lingua Franca file with one reactor is this:

```
target TypeScript
main reactor {
    reaction(startup) {=
        console.log("Hello World.")
    =}
}
```

Every Lingua Franca program begins with a [target declaration](#) that specifies the language in which reactions are written. This is also the language of the program(s) generated by the Lingua Franca code generator.

Every LF program also has a **main** or **federated** reactor, which is the top level of a hierarchy of contained and interconnected reactors. The above simple example has no contained reactors.

The **main** reactor above has a single **reaction**, which is triggered by the **startup** trigger. This trigger causes the reaction to execute at the start of the program. The body of the reaction, delimited by `{= ... =}`, is ordinary TypeScript code which, as we will see, has access to a number of functions and variables specific to Lingua Franca.

Examples

Examples of Lingua Franca programs can be found in [the examples-lingua-franca repository](#).

The [regression tests](#) have a rich set of examples that illustrate every feature of the language.

Structure of an LF Project

The Lingua Franca tools assume that LF programs are put into a file with a `.lf` extension that is stored somewhere within a directory called `src`. To compile and run the above example, choose a **project root** directory, create a `src` directory within that, and put the above code into a file

called, say, `src/HelloWorld.lf` . You can compile the code on the [command line](#), within [Visual Studio Code](#), or within the [Epoch IDE](#). On the command line this will look like this:

```
> lfc src/HelloWorld.lf
... output from the code generator and compiler ...
```

After this completes, an additional `src-gen` directory will have been created within the project root. The generated code will be in subdirectory called `HelloWorld` within `src-gen` . The output from the code generator will include instructions for executing the generated code:

```
#################################
To run the generated program, use:

    node ...path-to-project.../src-gen/Minimal/dist/Minimal.js

#################################
```

Reactor Block

A **reactor** is a software component that reacts to input events, timer events, and internal events. It has private state variables that are not visible to any other reactor. Its reactions can consist of altering its own state, sending messages to other reactors, or affecting the environment through some kind of actuation or side effect (e.g., printing a message, as in the above `HelloWorld` example).

The general structure of a reactor definition is as follows:

```
[main or federated] reactor <class-name> [(<parameters>)] {
    input <name>:<type>
    output <name>:<type>
    state <name>:<type>(<value>)
    timer <name>([<offset>, [<period>]])
    logical action <name>[:<type>]
    physical action <name>[:<type>]
    reaction(<triggers>) [<uses>] [=> <effects>] {= ... body ...=}
    <instance-name> = new <class-name>([<parameter-assignments>])
    <instance-name> [, ...] => <instance-name> [, ...] [after <delay>]
}
```

Contents within square brackets are optional, contents within `<...>` are user-defined, and each line may appear zero or more times, as explained in the next pages. Parameters, inputs, outputs, timers, actions, and contained reactors all have names, and the names are required to be distinct from one another.

If the **reactor** keyword is preceded by **main** or **federated**, then this reactor will be instantiated and run by the generated code.

Any number of reactors may be defined in one file, and a **main** or **federated** reactor need not be given a name, but if it is given a name, then that name must match the file name.

Reactors may extend other reactors, inheriting their properties, and a file may import reactors from other files. If an imported LF file contains a **main** or **federated** reactor, that reactor is ignored (it will not be imported). This makes it easy to create a library of reusable reactors that each come with a test case or demonstration in the form of a main reactor.

Comments

Lingua Franca files can have C/C++/Java-style comments and/or Python-style comments. All of the following are valid comments:

```
// Single-line C-style comment.
/*
 * Multi-line C-style comment.
 */
# Single-line Python-style comment.
```

Inputs and Outputs

In this section, we will endow reactors with inputs and outputs.

Input and Output Declarations

Input and output declarations have the form:

For example, the following reactor doubles its input and sends the result to the output:

```
target TypeScript
reactor Double {
    input x:number
    output y:number
    reaction(x) -> y {=
        y = value * 2
    =}
}
```

Notice how the input value is accessed and how the output value is set. This is done differently for each target language. See the [Target Language Details](#) for detailed documentation of these mechanisms. Setting an output within a reaction will trigger downstream reactions at the same [Logical Time](#) that the reaction is invoked (or, more precisely, at the same [tag](#)). If a particular output port is set more than once at any tag, the last set value will be the one that downstream reactions see. Since the order in which reactions of a reactor are invoked at a logical time is deterministic, and whether inputs are present depends only on their timestamps, the final value set for an output will also be deterministic.

The **reaction** declaration above indicates that an input event on port `x` is a **trigger** and that an output event on port `y` is a (potential) **effect**. A reaction can declare more than one trigger or effect by just listing them separated by commas. For example, the following reactor has two triggers and tests each input for presence before using it:

```
target TypeScript
reactor Destination {
    input x:number
    input y:number
    reaction(x, y) {=
        let sum = 0
        if (x !== undefined) {
            sum += x
        }
        if (y !== undefined) {
            sum += y
        }
        console.log(`Received ${sum}.`)
    =}
}
```

NOTE: if a reaction fails to test for the presence of an input and reads its value anyway, then the result it will get is target dependent. In the TS target, the value will be **undefined** if the input is not present, a legitimate value in TypeScript.

Triggers, Effects, and Uses

The general form of a **reaction** is

```
reaction (<triggers>) <uses> -> <effects> {=
    <target language code>
=}
```

The **triggers** field can be a comma-separated list of input ports, output ports of contained reactors, timers, actions, or the special events **startup** and **shutdown**. There must be at least one trigger for each reaction. A reaction with a **startup** trigger is invoked when the program begins executing, and a reaction with a **shutdown** trigger is invoked at the end of execution.

The **uses** field, which is optional, specifies input ports (or output ports of contained reactors) that do not trigger execution of the reaction but may be read by the reaction.

The **effects** field, which is also optional, is a comma-separated lists of output ports ports, input ports of contained reactors, or actions.

Setting an Output Multiple Times

If one or more reactions set an output multiple times at the same [tag](), then only the last value set will be seen by any downstream reactors.

If a reaction wishes to test whether an output has been previously set at the current tag by some other reaction, it can test it in the same way it tests inputs for presence.

Mutable Inputs

Normally, a reaction does not modify the value of an input. An input is said to be **immutable**. The degree to which this is enforced varies by target language. Most of the target languages make it rather difficult to enforce, so the programmer needs to avoid modifying the input. Modifying an input value may lead to nondeterministic results.

Occasionally, it is useful to modify an input. For example, the input may be a large data structure, and a reaction may wish to make a small modification and forward the result to an output. To accomplish this, the programmer should declare the input **mutable** as follows:

mutable input <name>:<type>;
mutable input <name>;

This is a directive to the code generator indicating that reactions that read this input may also modify the value of the input. The code generator will attempt to optimize the scheduling to avoid copying the input value, but this may not be possible, in which case it will automatically insert a copy operation, making it safe to modify the input. The target-specific reference documentation has more details about how this works.

Parameters and State Variables

Parameter Declaration

A reactor class definition can parameterized as follows:

For example, `{= int | null =}` defines nullable integer type in TypeScript.

Each parameter must have a *default value*, written `(<expr>)`. An expression may be a numeric constant, a string enclosed in quotation marks, a time value such as `10 msec`, a list of values, or target-language code enclosed in `{= ... =}`, for example. See [Expressions](#) for full details on what expressions are valid.

For example, the `Double` reactor on the [previous page](#) can be replaced with a more general parameterized reactor `Scale` as follows:

```
target TypeScript;
reactor Scale(factor:number(2)) {
    input x:number;
    output y:number;
    reaction(x) -> y {=
        if (x !== undefined) y = x * factor
    =}
}
```

This reactor, given any input event `x` will produce an output `y` with value equal to the input scaled by the `factor` parameter. The default value of the `factor` parameter is 2, but this can be changed when the `Scale` reactor is instantiated.

Notice how, within the body of a reaction, the code accesses the parameter value. This is different for each target language.

State Declaration

A reactor declares a state variable as follows:

The `<value>` is an initial value and, like parameter values, can be given as an [expression](#) or target language code with delimiters `{= ... =}`. The initial value can also be given as a parameter name. The value can be accessed and modified in a target-language-dependent way as illustrated by the following example:

```
target TypeScript
reactor Count {
    state count:number(0)
    output y:number
    timer t(0, 100 msec)
    reaction(t) -> y {=
        y = count++
    =}
}
```

This reactor has an integer state variable named `count`, and each time its reaction is invoked, it outputs the value of that state variable and increments it. The reaction is triggered by a **timer**, discussed in the next section.

Time and Timers

Logical Time

A key property of Lingua Franca is **logical time**. All events occur at an instant in logical time. By default, the runtime system does its best to align logical time with **physical time**, which is some measurement of time on the execution platform. The **lag** is defined to be physical time minus logical time, and the goal of the runtime system is maintain a small non-negative lag.

The **lag** is allowed to go negative only if the [`fast` target property](#) or the [--fast](#) is set to `true`. In that case, the program will execute as fast as possible with no regard to physical time.

Time Values

A time value is given with units (unless the value is 0, in which case the units can be omitted). The allowable units are:

- For nanoseconds: `ns`, `nsec`, or `nsecs`

- For microseconds: `us`, `usec`, or `usecs`

- For milliseconds: `ms`, `msec`, or `msecs`

- For seconds: `s`, `sec`, `secs`, `second`, or `seconds`

- For minutes: `min`, `minute`, `mins`, or `minutes`

- For hours: `h`, `hour`, or `hours`

- For days: `d`, `day`, or `days`

- For weeks: `week` or `weeks`

The following example illustrates using time values for parameters and state variables:

```
target TypeScript
main reactor SlowingClock(start:time(100 msec), incr:time(100 msec)) {
    state interval:time(start)
    logical action a
    reaction(startup) -> a {=
        actions.a.schedule(start, null);
    =}
    reaction(a) -> a {=
        console.log(`Logical time since start: ${util.getElapsedLogicalTime
        interval = interval.add(incr)
        actions.a.schedule(interval, null)
    =}
}
```

This has two time parameters, `start` and `incr`, each with default value `100 msec`. This parameter is used to initialize the `interval` state variable, which also stores a time. The **logical action** `a`, explained [below](#), is used to schedule events to occur at time `start` after program startup and then at intervals that are increased each time by `incr`. The result of executing this program will look like this:

```
Logical time since start: 100000000 nsec.
Logical time since start: 300000000 nsec.
Logical time since start: 600000000 nsec.
Logical time since start: 1000000000 nsec.
...
```

Timers

The simplest use of logical time in Lingua Franca is to invoke a reaction periodically. This is done by first declaring a **timer** using this syntax:

```
timer <name>(<offset>, <period>);
```

The `<period>`, which is optional, specifies the time interval between timer events. The `<offset>`, which is also optional, specifies the (logical) time interval between when the program starts executing and the first timer event. If no period is given, then the timer event occurs only once. If neither an offset nor a period is specified, then one timer event occurs at program start, simultaneous with the **startup** event.

The period and offset are given by a number and a units, for example, `10 msec`. See the [expressions documentation](#) for allowable units. Consider the following example:

```
target TypeScript
main reactor Timer {
    timer t(0, 1 sec)
    reaction(t) {=
        console.log(`Logical time is ${util.getCurrentLogicalTime()}.`)
    =}
}
```

This specifies a timer named `t` that will first trigger at the start of execution and then repeatedly trigger at intervals of one second. Notice that the time units can be left off if the value is zero.

Each target provides a built-in function for retrieving the logical time at which the reaction is invoked, util.getCurrentLogicalTime() . On most platforms (with the exception of some embedded platforms), the returned value is a 64-bit number representing the number of nanoseconds that have elapsed since January 1, 1970. Executing the above displays something like the following:

```
Logical time is 1648402121312985000.
Logical time is 1648402122312985000.
Logical time is 1648402123312985000.
...
```

The output lines appear at one second intervals unless the `fast` option has been specified.

Elapsed Time

The times above are a bit hard to read, so, for convenience, each target provides a built-in function to retrieve the *elapsed* time. For example:

```
target TypeScript
main reactor TimeElapsed {
    timer t(0, 1 sec)
    reaction(t) {=
        console.log(`Elapsed logical time is ${util.getElapsedLogicalTime()
    =}
}
```

See the [Target Language Details](#) for the full set of functions provided for accessing time values.

Executing this program will produce something like this:

```
Elapsed logical time is 0.
Elapsed logical time is 1000000000.
Elapsed logical time is 2000000000.
...
```

Comparing Logical and Physical Times

The following program compares logical and physical times:

```
target TypeScript
main reactor TimeLag {
    timer t(0, 1 sec)
    reaction(t) {=
        const t = util.getElapsedLogicalTime()
        const T = util.getElapsedPhysicalTime()
        console.log(`Elapsed logical time: ${t}, physical time: ${T}, lag:
    =}
}
```

Execution will show something like this:

```
Elapsed logical time: 0, physical time: 855000, lag: 855000
Elapsed logical time: 1000000000, physical time: 1004714000, lag: 4714000
Elapsed logical time: 2000000000, physical time: 2004663000, lag: 4663000
Elapsed logical time: 3000000000, physical time: 3000210000, lag: 210000
...
```

In this case, the lag varies from a few hundred microseconds to a small number of milliseconds. The amount of lag will depend on the execution platform.

Simultaneity and Instantaneity

If two timers have the same *offset* and *period*, then their events are logically simultaneous. No observer will be able to see that one timer has triggered and the other has not.

A reaction is always invoked at a well-defined logical time, and logical time does not advance during its execution. Any output produced by the reaction will be **logically simultaneous** with the input. In other words, reactions are **logically instantaneous** (for an exception, see Logical Execution Time). Physical time, however, does elapse during execution of a reaction.

Timeout

By default, a Lingua Franca program will terminate when there are no more events to process. If there is a timer with a non-zero period, then there will always be more events to process, so the default execution will be unbounded. To specify a finite execution horizon, you can either specify a `timeout` target property or a `--timeout` command-line option. For example, the following

`timeout` property will cause the above timer with a period of one second to terminate after 11 events:

```
target TypeScript {
    timeout: 10 sec
}
```

Startup and Shutdown

To cause a reaction to be invoked at the start of execution, a special **startup** trigger is provided:

```
reactor Foo {
    reaction(startup) {=
        ... perform initialization ...
    =}
}
```

The **startup** trigger is equivalent to a timer with no *offset* or *period*.

To cause a reaction to be invoked at the end of execution, a special **shutdown** trigger is provided. Consider the following reactor, commonly used to build regression tests:

```
target TypeScript
reactor TestCount(start:number(0), stride:number(1), numInputs:number(1)) {
    state count:number(start)
    state inputsReceived:number(0)
    input x:number
    reaction(x) {=
        console.log(`Received ${x}`)
        if (x != count) {
            console.error(`ERROR: Expected ${count}.`)
            process.exit(1)
        }
        count += stride;
        inputsReceived++
    =}
    reaction(shutdown) {=
        console.log("Shutdown invoked.")
        if (inputsReceived != numInputs) {
            console.error(`ERROR: Expected to receive ${numInputs}, but got
            process.exit(2)
        }
    =}
}
```

This reactor tests its inputs against expected values, which are expected to start with the value given by the `start` parameter and increase by `stride` with each successive input. It expects to receive a total of `num_inputs` input events. It checks the total number of inputs received in its **shutdown** reaction.

The **shutdown** trigger typically occurs at [microstep](#) 0, but may occur at a larger microstep. See [Superdense Time](#) and [Termination](#).

Composing Reactors

Contained Reactors

Reactors can contain instances of other reactors defined in the same file or in an imported file. Assume the `Count` and `Scale` reactors defined in [Parameters and State Variables](#) are stored in files `Count.lf` and `Scale.lf`, respectively, and that the `TestCount` reactor from [Time and Timers](#) is stored in `TestCount.lf`. Then the following program composes one instance of each of the three:

```
target TypeScript {
    timeout: 1 sec,
    fast: true
}
import Count from "Count.lf"
import Scale from "Scale.lf"
import TestCount from "TestCount.lf"

main reactor RegressionTest {
    c = new Count()
    s = new Scale(factor = 4)
    t = new TestCount(stride = 4, numInputs = 11)
    c.y -> s.x
    s.y -> t.x
}
```

Diagrams

As soon as programs consist of more than one reactor, it becomes particularly useful to reference the diagrams that are automatically created and displayed by the Lingua Franca IDEs. The diagram for the above program is as follows:

![Lingua Franca diagram]

In this diagram, the timer is represented by a clock-like icon, the reactions by chevron shapes, and the **shutdown** event by a diamond. If there were a **startup** event in this program, it would appear as a circle.

Creating Reactor Instances

An instance is created with the syntax:

```
<instance_name> = new <class_name>(<parameters>)
```

A bank with several instances can be created in one such statement, as explained in the [banks of reactors documentation](#).

The `<parameters>` argument is a comma-separated list of assignments:

```
<parameter_name> = <value>, ...
```

Like the default value for parameters, `<value>` can be a numeric constant, a string enclosed in quotation marks, a time value such as `10 msec`, target-language code enclosed in `{= ... =}`, or any of the list forms described in [Expressions](#).

Connections

Connections between ports are specified with the syntax:

```
<source_port_reference> -> <destination_port_reference>
```

where the port references are either `<instance_name>.<port_name>` or just `<port_name>`, where the latter form is used for connections that cross hierarchical boundaries, as illustrated in the next section.

On the left and right of a connection statement, you can put a comma-separated list. For example, the above pair of connections can be written,

```
c.y, s.y -> s.x, t.x
```

The only constraint is that the total number of channels on the left match the total number on the right.

A destination port (on the right) can only be connected to a single source port (on the left). However, a source port may be connected to multiple destinations, as in the following example:

Lingua Franca diagram

Lingua Franca provides a convenient shortcut for such multicast connections, where the above two lines can be replaced by one as follows:

```
(a.y)+ -> b1.x, b2.x
```

The enclosing `( ... )+` means to repeat the enclosed comma-separated list of sources however many times is needed to provide inputs to all the sinks on the right of the connection `->`.

Import Statement

An import statement has the form:

```
import <classname> as <alias> from "<path>"
```

where `<classname>` and `<alias>` can be a comma-separated list to import multiple reactors from the same file. The `<path>` specifies another `.lf` file relative to the location of the current file. The `as <alias>` portion is optional and specifies alternative class names to use in the **new** statements.

Hierarchy

Reactors can be composed in arbitrarily deep hierarchies. For example, the following program combines the `Count` and `Scale` reactors within on `Container`:

```
target TypeScript
import Count from "Count.lf"
import Scale from "Scale.lf"
import TestCount from "TestCount.lf"

reactor Container(stride:number(2)) {
    output y:number
    c = new Count()
    s = new Scale(factor = stride)
    c.y -> s.x
    s.y -> y
}
main reactor Hierarchy {
    c = new Container(stride = 4)
    t = new TestCount(stride = 4, numInputs = 11)
    c.y -> t.x
}
```

Lingua Franca diagram

The `Container` has a parameter named `stride`, whose value is passed to the `factor` parameter of the `Scale` reactor. The line

```
s.y -> y;
```

establishes a connection across levels of the hierarchy. This propagates the output of a contained reactor to the output of the container. A similar notation may be used to propagate the input of a container to the input of a contained reactor,

```
x -> s.x;
```

Connections with Logical Delays

Connections may include a **logical delay** using the **after** keyword, as follows:

```
<source_port_reference> -> <destination_port_reference> after <time_val
```

where `<time_value>` can be any of the forms described in Expressions.

The **after** keyword specifies that the logical time of the event delivered to the destination port will be larger than the logical time of the reaction that wrote to source port. The time value is required to be non-negative, but it can be zero, in which case the input event at the receiving end will be one [microstep](#) later than the event that triggered it.

Physical Connections

A subtle and rarely used variant of the `->` connection is a **physical connection**, denoted `~>`. For example:

```
main reactor {
    a = new A();
    b = new B();
    a.y ~> b.x;
}
```

This is rendered in by the diagram synthesizer as follows:

Lingua Franca diagram

In such a connection, the logical time at the recipient is derived from the local physical clock rather than being equal to the logical time at the sender. The physical time will always exceed the logical time of the sender (unless fast is set to `true`), so this type of connection incurs a nondeterministic positive logical time delay. Physical connections are useful sometimes in [Distributed-Execution](#) in situations where the nondeterministic logical delay is tolerable. Such connections are more efficient because timestamps need not be transmitted and messages do not need to flow through through a centralized coordinator (if a centralized coordinator is being used).

Reactions and Methods

Reaction Order

A reactor may have multiple reactions, and more than one reaction may be enabled at any given tag. In Lingua Franca semantics, if two or more reactions of the same reactor are **simultaneously enabled**, then they will be invoked sequentially in the order in which they are declared. More strongly, the reactions of a reactor are **mutually exclusive** and are invoked in tag order primarily and declaration order secondarily. Consider the following example:

```
target TypeScript {
    timeout: 3s
}
main reactor Alignment {
    state s:number(0)
    timer t1(100ms, 100ms)
    timer t2(200ms, 200ms)
    timer t4(400ms, 400ms)
    reaction(t1) {=
        s += 1
    =}
    reaction(t2) {=
        s -= 2
    =}
    reaction(t4) {=
        console.log(`s = ${s}`)
    =}
}
```

Every 100 ms, this increments the state variable `s` by 1, every 200 ms, it decrements `s` by 2, and every 400 ms, it prints the value of `s`. When these reactions align, they are invoked in declaration order, and, as a result, the printed value of `s` is always 0.

Overwriting Outputs

Just as the reactions of the `Alignment` reactor overwrite the state variable `s`, logically simultaneous reactions can overwrite outputs. Consider the following example:

```
target TypeScript
reactor Overwriting {
    output y:number
    state s:number(0)
    timer t1(100 msec, 100 msec)
    timer t2(200 msec, 200 msec)
    reaction(t1) -> y {=
        s += 1
        y = s
    =}
    reaction(t2) -> y {=
        s -= 2
        y = s
    =}
}
```

Here, the reaction to `t1` will set the output to 1 or 2, but every time it sets it to 2, the second reaction (to `t2`) will overwrite the output with the value 0. As a consequence, the outputs will be 1, 0, 1, 0, ... deterministically.

Reacting to Outputs of Contained Reactors

A reaction may be triggered by the an input to the reactor, but also by an output of a contained reactor, as illustrated in the following example:

```
target TypeScript
import Overwriting from "Overwriting.lf";
main reactor {
    s = new Overwriting();
    reaction(s.y) {=
        if (s.y != 0 && s.y != 1) {
            util.requestErrorStop("Outputs should only be 0 or 1!")
        }
    =}
}
```

This instantiates the above `Overwriting` reactor and monitors its outputs.

Method Declaration

Causality Loops

Cycles

The interconnection pattern for a collection of reactors can form a cycle, but some care is required. Consider the following example:

```
target TypeScript
reactor A {
    input x:number
    output y:number
    reaction(x) -> y {=
        // ... something here ...
    =}
}
reactor B {
    input x:number
    output y:number
    reaction(x) {=
        // ... something here ...
    =}
    reaction(startup) -> y {=
        // ... something here ...
    =}
}
main reactor {
    a = new A()
    b = new B()
    a.y -> b.x
    b.y -> a.x
}
```

This program yields the following diagram:

![Lingua Franca diagram]

The diagram highlights a **causality loop** in the program. At each tag, in reactor `B` , the first reaction has to execute before the second if it is enabled, a precedence indicated with the red dashed arrow. But the first can't execute until the reaction of `A` has executed, and that reaction cannot execute until the second reaction `B` has executed. There is no way to satisfy these requirements, so the tools refuse to generated code.

Cycles with Delays

One way to break the causality loop and get an executable program is to introduce a [logical delay](#) into the loop, as shown below:

```
target TypeScript
reactor A {
    input x:number
    output y:number
    reaction(x) -> y {=
        // ... something here ...
    =}
}
reactor B {
    input x:number
    output y:number
    reaction(x) {=
        // ... something here ...
    =}
    reaction(startup) -> y {=
        // ... something here ...
    =}
}
main reactor {
    a = new A()
    b = new B()
    a.y -> b.x after 0
    b.y -> a.x
}
```


Lingua Franca diagram

Here, we have used a delay of 0, which results in a delay of one [microstep](#). We could equally well have specified a positive time value.

Reaction Order

Frequently, a program will have such cycles, but you don't want a logical delay in the loop. To get a cycle without logical delays, the reactions need to be reordered, as shown below:

```
target TypeScript
reactor A {
    input x:number
    output y:number
    reaction(x) -> y {=
        // ... something here ...
    =}
}
reactor B {
    input x:number
    output y:number
    reaction(startup) -> y {=
        // ... something here ...
    =}
    reaction(x) {=
        // ... something here ...
    =}
}
main reactor {
    a = new A()
    b = new B()
    a.y -> b.x
    b.y -> a.x
}
```

There is no longer any causality loop.

Extending Reactors

Extending a Base Reactor

Actions

Action Declaration

An action declaration has one of the following forms:

```
logical action <name>(<min_delay>, <min_spacing>, <policy>)
physical action <name>(<min_delay>, <min_spacing>, <policy>)
```

The `min_delay` , `min_spacing` , and `policy` are all optional. If only one argument is given in parentheses, then it is interpreted as an `min_delay` , if two are given, then they are interpreted as `min_delay` and `min_spacing` . The `min_delay` and `min_spacing` are time values. The `policy` argument is a string that can be one of the following: `"defer"` (the default), `"drop"` , or `"replace"` . Note that the quotation marks are needed.

Logical Actions

Timers are useful to trigger reactions once or periodically. Actions are used to trigger reactions more irregularly. An action, like an output or input port, can carry data, but unlike a port, an action is visible only within the reactor that defines it.

There are two kinds of actions, **logical** and **physical**. A **logical action** is used by a reactor to schedule a trigger at a fixed logical time interval d into the future. The time interval d, which is called a **delay**, is relative to the logical time t at which the scheduling occurs. If a reaction executes at logical time t and schedules an action `a` with delay d, then any reaction that is triggered by `a` will be invoked at logical time $t + d$. For example, the following reaction schedules something (printing the current elapsed logical time) 200 msec after an input `x` arrives:

```
target TypeScript
reactor Schedule {
    input x:number
    logical action a
    reaction(x) -> a {=
        actions.a.schedule(TimeValue.nsecs(200), null)
    =}
    reaction(a) {=
        console.log(`Action triggered at logical time ${util.getElapsedLogi
    =}
}
```

Lingua Franca diagram

Here, the delay is specified in the call to schedule within the target language code. Notice that in the diagram, a logical action is shown as a triangle with an **L**. Logical actions are always scheduled within a reaction of the reactor that declares the action.

The time argument is required to be non-negative. If it is zero, then the action will be scheduled one **microstep** later. See Superdense Time.

The `schedule()` method of an action takes two arguments, a `TimeValue` and an (optional) payload. If a payload is given and a type is given for the action, then the type of the payload must match the type of the action. See the Target Language Details for details.

Physical Actions

A **physical action** is used to schedule reactions at logical times determined by the local physical clock. If a physical action with delay *d* is scheduled at *physical* time *T*, then the *logical time* assigned to the event is *T + d*. For example, the following reactor schedules the physical action `p` to trigger at a **logical time** equal to the **physical time** at which the input `x` arrives:

```
target TypeScript
reactor Physical {
    input x:int
    physical action a
    reaction(x) -> a {=
        actions.a.schedule(TimeValue.zero(), null)
    =}
    reaction(a) {=
        console.log(`Action triggered at logical time ${util.getElapsedLogi
    =}
}
```

Lingua Franca diagram

If you drive this with a timer, using for example the following structure:

Lingua Franca diagram

then running the program will yield an output something like this:

```
Action triggered at logical time 201491000 nsec after start.
Action triggered at logical time 403685000 nsec after start.
Action triggered at logical time 603669000 nsec after start.
...
```

Here, logical time is lagging physical time by a few milliseconds. Note that, unless the fast option is given, logical time t chases physical time T, so $t < T$. Hence, the event being scheduled in the reaction to input `x` is assured of being in the future in logical time.

Whereas logical actions are required to be scheduled within a reaction of the reactor that declares the action, physical actions can be scheduled by code that is outside the Lingua Franca system. For example, some other thread or a callback function may call `schedule()`, passing it a physical action. For example:

```
target TypeScript
main reactor {

    physical action a(100 msec):number;

    reaction(startup) -> a {=
        // Have asynchronous callback schedule physical action.
        setTimeout(() => {
            actions.a.schedule(TimeValue.zero(), 0)
        }, 200)
    =}

    reaction(a) {=
        console.log(`Action triggered at logical time ${util.getElapsedLogi
    =}
}
```

Lingua Franca diagram

Physical actions are the mechanism for obtaining input from the outside world. Because they are assigned a logical time derived from the physical clock, their logical time can be interpreted as a measure of the time at which some external event occurred.

Triggering Time for Actions

An action will trigger at a logical time that depends on the arguments given to the schedule function, the `<min_delay>`, `<min_spacing>`, and `<policy>` arguments in the action declaration, and whether the action is physical or logical.

For a **logical** action `a`, the tag assigned to the event resulting from a call to `schedule()` is computed as follows. First, let *t* be the *current logical time*. For a logical action, *t* is just the logical time at which the reaction calling `schedule()` is called. The **preliminary time** of the action is then just *t* + `<min_delay>` + `<offset>`. This preliminary time may be further modified, as explained below.

For a **physical** action, the preliminary time is similar, except that *t* is replaced by the current *physical* time *T* when `schedule()` is called.

If a `<min_spacing>` has been declared, then it gives a minimum logical time interval between the tags of two subsequently scheduled events. If the preliminary time is closer than `<min_spacing>` to the time of the previously scheduled event (if there is one), then `<policy>` (if supported by the target) determines how the minimum spacing constraint is enforced.

Testing an Action for Presence

When a reaction is triggered by more than one action or by an action and an input, it may be necessary to test within the reaction whether the action is present. Just like for inputs, this can be done in the TypeScript target with `a != undefined`, where `a` is the name of the action.

Superdense Time

Tag vs. Time

The model of time in Lingua Franca is a bit more sophisticated than we have hinted at. Specifically, a **superdense** model of time is used. In particular, instead of a **timestamp**, LF uses a **tag**, which consists of a **logical time** *t* and a **microstep** *m*.

A [logical action](#) may have a `<min_delay>` of zero, and the `<offset>` argument to the `schedule()` function may be zero. In this case, the call to `schedule()` appears to be requesting that the action trigger at the *current logical time*. Here is where superdense time comes in. The action will indeed trigger at the current logical time, but one microstep later. Consider the following example:

```
target TypeScript
main reactor {
    state count:number(1)
    logical action a
    reaction(startup, a) -> a {=
        console.log(`${count}. Logical time is ${util.getCurrentLogicalTime
        if (count++ < 5) {
            actions.a.schedule(TimeValue.zero(), null)
        }
    =}
}
```


Lingua Franca diagram

Executing this program will yield something like this:

```
1. Logical time is 1649607749415269000. Microstep is 0.
2. Logical time is 1649607749415269000. Microstep is 1.
3. Logical time is 1649607749415269000. Microstep is 2.
4. Logical time is 1649607749415269000. Microstep is 3.
5. Logical time is 1649607749415269000. Microstep is 4.
```

Notice that the logical time is not advancing, but the microstep is (the logical time, in this case, gives the number of nanoseconds that have elapsed since January 1, 1970). The general rule is that **every** call to `schedule()` advances the tag by at least one microstep.

Logical Simultaneity

Two events are **logically simultaneous** only if *both* the logical time and the microstep are equal. The following example illustrates this:

```
target TypeScript
reactor Destination {
    input x:number
    input y:number
    reaction(x, y) {=
        console.log(`Time since start: ${util.getElapsedLogicalTime()}, mic
        if (x !== undefined) {
            console.log("  x is present.")
        }
        if (y !== undefined) {
            console.log("  y is present.")
        }
    =}
}
main reactor {
    logical action repeat
    d = new Destination()
    reaction(startup) -> d.x, repeat {=
        d.x = 1
        actions.repeat.schedule(0, null)
    =}
    reaction(repeat) -> d.y {=
        d.y = 1
    =}
}
```

The `Destination` reactor has two inputs, `x` and `y`, and it reports in a reaction to either input what is the logical time, the microstep, and which input is present. The main reactor reacts to **startup** by sending data to the `x` input of `Destination`. It then schedules a `repeat` action with an `<offset>` of zero. The `repeat` reaction is invoked **strictly later**, one **microstep** later. The output printed, therefore, will look like this:

```
Time since start: 0, microstep: 0
  x is present.
Time since start: 0, microstep: 1
  y is present.
```

The reported elapsed logical time has not advanced in the second reaction, but the fact that `x` is not present in the second reaction proves that the first reaction and the second are not logically simultaneous. The second occurs one microstep later.

Alignment of Logical and Physical Times

Recall that in Lingua Franca, logical time "chases" physical time, invoking reactions at a physical time close to their logical time. For that purpose, the microstep is ignored.

Modal Reactors

Deadlines

Lingua Franca includes a notion of a **deadline**, which is a constraint on the relation between logical time and physical time. Specifically, a program may specify that the invocation of a reaction must occur within some *physical* time interval of the *logical* time of the message. If a reaction is invoked at logical time 12 noon, for example, and the reaction has a deadline of one hour, then the reaction is required to be invoked before the physical-time clock of the execution platform reaches 1 PM. If the deadline is violated, then the specified deadline handler is invoked instead of the reaction.

Purposes for Deadlines

A deadline in an LF program serves two purposes. First, it can guide scheduling in that a scheduler may prioritize reactions with deadlines over those without or those with longer deadlines. For this purpose, if a reaction has a deadline, then all upstream reactions on which it depends (without logical delay) inherit its deadline. Hence, those upstream reactions will also be given higher priority.

Second, the deadline mechanism provides a **fault handler**, a section of code to invoke when the deadline requirement is violated. Because invocation of the fault handler depends on factors beyond the control of the LF program, an LF program with deadlines becomes **nondeterministic**. The behavior of the program depends on the exact timing of the execution.

There remains the question of when the fault handler should be invoked. By default, deadlines in LF are **lazy**, meaning that the fault handler is invoked at the logical time of the event triggering the reaction whose deadline is missed. Specifically, the possible violation of a deadline is not checked until the reaction with the deadline is ready to execute. Only then is the determination made whether to invoke the regular reaction or the fault handler.

An alternative is an **eager deadline**, where a fault handler is invoked as soon as possible after a deadline violation becomes inevitable. With an eager deadline, if an event with tag (t, m) triggers a reaction with deadline D, then as soon as the runtime system detects that physical time $T > t + D$, the fault handler becomes enabled. This can occur at a logical time *earlier* than t. Hence, a fault handler may be invoked at a logical time earlier than that of the event that triggered the fault.

Note: As of this writing, eager deadlines are not implemented in any LF target language, so all deadlines are lazy.

Lazy Deadline

A lazy deadline is specified as follows:

```
target TypeScript
reactor Deadline {
    input x:number
    output d:number // Produced if the deadline is violated.
    reaction(x) -> d {=
        console.log("Normal reaction.")
    =} deadline(10 msec) {=
        console.log("Deadline violation detected.")
        d = x
    =}
}
```

This reactor specifies a deadline of 10 milliseconds (this can be a parameter of the reactor). If the reaction to `x` is triggered later in physical time than 10 msec past the timestamp of `x`, then the second body of code is executed instead of the first. That second body of code has access to anything the first body of code has access to, including the input `x` and the output `d`. The output can be used to notify the rest of the system that a deadline violation occurred. This reactor can be tested as follows:

```
target TypeScript
import Deadline from "Deadline.lf"
main reactor {
    logical action a
    d = new Deadline()
    reaction(startup) -> d.x, a {=
        d.x = 0
        actions.a.schedule(TimeValue.zero(), null)
    =}
    reaction(a) -> d.x {=
        d.x = 0
        for (const later = util.getCurrentPhysicalTime().add(TimeValue.msec
            util.getCurrentPhysicalTime() < later;) {
            // Take time...
        }
    =}
    reaction(d.d) {=
        console.log("Deadline reactor produced an output.")
    =}
}
```

![Lingua Franca diagram]

Running this program will result in the following output:

```
Normal reaction.
Deadline violation detected.
Deadline reactor produced an output.
```

The first reaction of the `Deadline` reactor does not violate the deadline, but the second does. Notice that the sleep in the **main** reactor occurs *after* setting the output, but because of the deterministic semantics of LF, this does not matter. The actual value of an output cannot be known until every reaction that sets that output *completes* its execution. Since this reaction takes at least 20 msec to complete, the deadline is assured of being violated.

Notice that the deadline is annotated in the diagram with a small clock symbol.

Deadline Violations During Execution

Whether a deadline violation occurs is checked only *before* invoking the reaction with a deadline. What if the reaction itself runs for long enough that the deadline gets violated *during* the reaction

execution? For this purpose, a target-language function is provided to check whether a deadline is violated during execution of a reaction with a deadline.

Consider this example:

```
WARNING: No source file found: ../code/ts/src/CheckDeadline.lf
```

Multiports and Banks

Lingua Franca provides a compact syntax for ports that can send or receive over multiple channels and another syntax for multiple instances of a reactor class. These are respectively called **multiports** and **banks of reactors**.

Multiports

To declare an input or output port to be a **multiport**, use the following syntax:

```
input[<width>] <name>
output[<width>] <name>
```

where `<width>` is a positive integer. This can be given either as an integer literal or a parameter name. Consider the following example:

```
target TypeScript
reactor Source {
    output[4] out:number
    reaction(startup) -> out {=
        for (let i = 0 ; i < out.length; i++) {
            out[i] = i
        }
    =}
}
reactor Destination {
    input[4] inp:number
    reaction(inp) {=
        let sum = 0
        for (let i = 0 ; i < inp.length; i++) {
            const val = inp[i]
            if (val) sum += val
        }
        console.log(`Sum of received: ${sum}`)
    =}
}
main reactor {
    a = new Source()
    b = new Destination()
    a.out -> b.inp
}
```


Lingua Franca diagram

Executing this program will yield:

```
Sum of received: 6.
```

The `Source` reactor has a four-way multiport output and the `Destination` reactor has a four-way multiport input. These channels are connected all at once on one line, the second line from the last. Notice that the generated diagram shows multiports with hollow triangles. Whether it shows the widths is controlled by an option in the diagram generator.

The `Source` reactor specifies `out` as an effect of its reaction using the syntax `-> out`. This brings into scope of the reaction body a way to access the width of the port and a way to write to each channel of the port.

NOTE: In `Destination`, the reaction is triggered by `in`, not by some individual channel of the multiport input. Hence, it is important when using multiport inputs to test for presence of the input on each channel, as done above with the syntax . An event on any one of the channels is sufficient to trigger the reaction.

Parameterized Widths

The width of a port may be given by a parameter. For example, the above `Source` reactor can be rewritten

```
reactor Source(width:int(4)) {
    output[width] out:int;
    reaction(startup) -> out {=
        ...
    =}
}
```

Connecting Reactors with Different Widths

Assume that the `Source` and `Destination` reactors above both use a parameter `width` to specify the width of their ports. Then the following connection is valid:

```
main reactor {
    a1 = new Source(width = 3);
    a2 = new Source(width = 2);
    b = new Destination(width = 5);
    a1.out, a2.out -> b.in;
}
```

The first three ports of `b` will received input from `a1`, and the last two ports will receive input from `a2`. Parallel composition can appear on either side of a connection. For example:

```
    a1.out, a2.out -> b1.out, b2.out, b3.out;
```

If the total width on the left does not match the total width on the right, then a warning is issued. If the left side is wider than the right, then output data will be discarded. If the right side is wider than the left, then input channels will be absent.

Any given port can appear only once on the right side of the `->` connection operator, so all connections to a multiport destination must be made in one single connection statement.

Banks of Reactors

Using a similar notation, it is possible to create a bank of reactors. For example, we can create a bank of four instances of `Source` and four instances of `Destination` and connect them as follows:

```
main reactor {
    a = new[4] Source();
    b = new[4] Destination();
    a.out -> b.in;
}
```


Lingua Franca diagram

If the `Source` and `Destination` reactors have multiport inputs and outputs, as in the examples above, then a warning will be issued if the total width on the left does not match the total width on the right. For example, the following is balanced:

```
main reactor {
    a = new[3] Source(width = 4);
    b = new[4] Destination(width = 3);
    a.out -> b.in;
}
```

There will be three instances of `Source`, each with an output of width four, and four instances of `Destination`, each with an input of width 3, for a total of 12 connections.

To distinguish the instances in a bank of reactors, the reactor can define a parameter called **bank_index**. If such a parameter is defined for the reactor, then when the reactor is instantiated in a bank, each instance will be assigned a number between 0 and n-1, where n is the number of reactor instances in the bank. For example, the following source reactor increments the output it produces by the value of `bank_index` on each reaction to the timer:

```
target TypeScript
reactor MultiportSource {
    timer t(0, 200 msec)
    output out:number
    state s:number(0)
    reaction(t) -> out {=
        out = s
        s += this.getBankIndex()
    =}
}
```

The width of a bank may also be given by a parameter, as in

```
main reactor(
    source_bank_width:int(3),
    destination_bank_width:int(4)
) {
    a = new[source_bank_width] Source(width = 4);
    b = new[destination_bank_width] Destination(width = 3);
    a.out -> b.in;
}
```
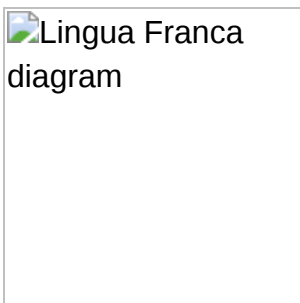
Contained Banks

Banks of reactors can be nested. For example, note the following program:

```
target TypeScript
reactor Child {
    reaction(startup) {=
        console.log(`My bank index ${this.getBankIndex()}`)
    =}
}
reactor Parent {
    c = new[2] Child()
}
main reactor {
    p = new[2] Parent()
}
```


Lingua Franca diagram

In this program, the `Parent` reactor contains a bank of `Child` reactor instances with a width of 2. In the main reactor, a bank of `Parent` reactors is instantiated with a width of 2, therefore, creating 4 `Child` instances in the program in total. The output of this program will be:

```
My bank index: 0.
My bank index: 1.
My bank index: 0.
My bank index: 1.
```

The order of these outputs will be nondeterministic if the execution is multithreaded (which it will be by default) because there is no dependence between the reactions, and, hence, they can execute in parallel.

The bank index of a container (parent) reactor can be passed down to contained (child) reactors. For example, note the following program:

```
target TypeScript
reactor Child (
    parentBankIndex:number(0)
) {
    reaction(startup) {=
        console.log(`My bank index: ${this.getBankIndex()} My parent's bank
    =}
}
reactor Parent {
    c = new[2] Child(parentBankIndex = {= this.getBankIndex() =})
}
main reactor {
    p = new[2] Parent()
}
```

In this example, the bank index of the `Parent` reactor is passed to the `parent_bank_index` parameter of the `Child` reactor instances. The output from this program will be:

```
My bank index: 1. My parent's bank index: 1.
My bank index: 0. My parent's bank index: 0.
My bank index: 0. My parent's bank index: 1.
My bank index: 1. My parent's bank index: 0.
```

Again, note that the order of these outputs is nondeterministic.

Finally, members of contained banks of reactors can be individually addressed in the body of reactions of the parent reactor if their input/output port appears in the reaction signature. For example, note the following program:

```
target TypeScript
reactor Child (
    parentBankIndex:number(0)
) {
    output out:number;
    reaction(startup) -> out {=
        out = parentBankIndex * 2 + this.getBankIndex()
    =}
}
reactor Parent {
    c = new[2] Child(parentBankIndex = {= this.getBankIndex() =})
    reaction(c.out) {=
        for (let i = 0; i < c.length; i++) {
            console.log(`Received ${c[i].out} from child ${i}`)
        }
    =}
}
main reactor {
    p = new[2] Parent();
}
```


Lingua Franca diagram

Running this program will give something like the following:

```
Received 0 from child 0.
Received 1 from child 1.
Received 2 from child 0.
Received 3 from child 1.
```

Note that that bank instance `c` in TypeScript is an array, so `c.length` is the width of the bank, and the bank members are referenced by indexing the array, as in `c[i]`.

Combining Banks and Multiports

Banks of reactors may be combined with multiports, as in the following example:

```
target TypeScript
reactor Source {
    output[3] out:number
    reaction(startup) -> out {=
        for (let i = 0 ; i < out.length; i++) {
            out[i] = i
        }
    =}
}
reactor Destination {
    input inp:number
    reaction(inp) {=
        console.log(`Destination ${this.getBankIndex()} received ${inp}`)
    =}
}

main reactor MultiportToBank {
    a = new Source()
    b = new[3] Destination()
    a.out -> b.inp
}
```


Lingua Franca diagram

The three outputs from the `Source` instance `a` will be sent, respectively, to each of three instances of `Destination`, `b[0]`, `b[1]`, and `b[2]`. The result of the program will be something like the following:

```
Destination 0 received 0.
Destination 1 received 1.
Destination 2 received 2.
```

Again, the order is nondeterministic in a multithreaded context.

The reactors in a bank may themselves have multiports. In all cases, the number of ports on the left of a connection must match the number on the right, unless the ones on the left are iterated, as explained next.

Broadcast Connections

Occasionally, you will want to have fewer ports on the left of a connection and have their outputs used repeatedly to broadcast to the ports on the right. In the following example, the outputs from an ordinary port are broadcast to the inputs of all instances of a bank of reactors:

```
reactor Source {
        output out:int;
        reaction(startup) -> out {=
                ... write to out ...
        =}
}
reactor Destination {
        input in:int;
        reaction(in) {=
                ... read from in ...
        =}
}
main reactor ThreadedThreaded(width:int(4)) {
        a = new Source();
        d = new[width] Destination();
        (a.out)+ -> d.in;
}
```

The syntax `(a.out)+` means "repeat the output port `a.out` one or more times as needed to supply all the input ports of `d.in`." The content inside the parentheses can be a comma-separated list of ports, the ports inside can be ordinary ports or multiports, and the reactors inside can be ordinary reactors or banks of reactors. In all cases, the number of ports inside the parentheses on the left must divide the number of ports on the right.

Interleaved Connections

Sometimes, we don't want to broadcast messages to all reactors, but need more fine-grained control as to which reactor within a bank receives a message. If we have separate source and destination reactors, this can be done by combining multiports and banks as was shown in [Combining Banks and Multiports](#). Setting a value on the index *n* of the output multiport, will result in a message to the *n*-th reactor instance within the destination bank. However, this pattern gets slightly more complicated, if we want to exchange addressable messages between instances of the same bank. This pattern is shown in the following example:

```
target TypeScript
reactor Node(numNodes: number(4)) {
    input[numNodes] inp: number
    output[numNodes] out: number

    reaction (startup) -> out {=
        out[1] = 42
        console.log(`Bank index ${this.getBankIndex()} sent 42 on channel 1
    =}

    reaction (inp) {=
        for (let i = 0; i < in.length; i++) {
            if (in[i] !== undefined) {
                console.log(`Bank index ${this.getBankIndex()} received ${i
            }
        }
    =}
}
main reactor(numNodes: number(4)) {
    nodes = new[numNodes] Node(numNodes=numNodes);
    nodes.out -> interleaved(nodes.inp)
}
```

Lingua Franca diagram

In the above program, four instance of `Node` are created, and, at startup, each instance sends 42 to its second (index 1) output channel. The result is that the second bank member (`bank_index` 1) will receive the number 42 on each input channel of its multiport input. Running this program gives something like the following:

```
Bank index 0 sent 42 on channel 1.
Bank index 1 sent 42 on channel 1.
Bank index 2 sent 42 on channel 1.
Bank index 3 sent 42 on channel 1.
Bank index 1 received 42 on channel 0.
Bank index 1 received 42 on channel 1.
Bank index 1 received 42 on channel 2.
Bank index 1 received 42 on channel 3.
```

In bank index 1, the 0-th channel receives from `bank_index` 0, the 1-th channel from `bank_index` 1, etc. In effect, the choice of output channel specifies the destination reactor in the bank, and the input channel specifies the source reactor from which the input comes.

This style of connection is accomplished using the new keyword **interleaved** in the connection. Normally, a port reference such as `nodes.out` where `nodes` is a bank and `out` is a multiport, would list all the individual ports by first iterating over the banks and then, for each bank index, iterating over the ports. If we consider the tuple (b,p) to denote the index b within the bank and the index p within the multiport, then the following list is created: (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3). However, if we use **interleaved** `(nodes.out)` instead, the connection logic will iterate over the ports first and then the banks, creating the following list: (0,0), (1,0), (2,0), (3,0), (0,1), (1,1), (2,1), (3,1), (0,2), (1,2), (2,2), (3,2), (0,3), (1,3), (2,3), (3,3). By combining a normal port reference with a interleaved reference, we can construct a fully connected network. The figure below visualizes this how this pattern would look without banks or multiports:

Lingua Franca diagram

If we were to use a normal connection `nodes.out -> nodes.in;` instead of the **interleaved** connection, then the following pattern would be created:

Lingua Franca diagram

Effectively, this connects each reactor instance to itself, which isn't very useful.

Generic Reactors

Preambles

Preamble

Reactions may contain arbitrary target-language code, but often it is convenient for that code to invoke external libraries or to share procedure definitions. For either purpose, a reactor may include a **preamble** section.

For example, the following reactor uses Node's built-in path module to extract the base name from a path:

```
target TypeScript;
main reactor Preamble {
    preamble {=
        import * as path from 'path';
    =}
    reaction (startup) {=
        var filename = path.basename('/Users/Refsnes/demo_path.js');
        console.log(filename);
    =}
}
```

This will print:

```
demo_path.js
```

By putting the `import` in the **preamble**, the library becomes available in all reactions of this reactor. Oddly, it also becomes available in all subsequently defined reactors in the same file. It's a bit more complicated to [set up Node.js modules from npm](#) that aren't built-in, but the reaction code to `import` them is the same as what you see here.

You can also use the preamble to define functions that are shared across reactions and reactors:

```
main reactor Preamble {
    preamble {=
        function add42( i:number) {
            return i + 42;
        }
    =}
    timer t;
    reaction(t) {=
        let s = "42";
        let radix = 10;
        let i = parseInt(s, radix);
        console.log("Converted string " + s + " to number " + i);
        console.log("42 plus 42 is " + add42(42));
    =}
}
```

Not surprisingly, this will print:

```
Converted string 42 to number 42
42 plus 42 is 84
```

Using Node Modules

Installing Node.js modules for TypeScript reactors with `npm` is essentially the same as installing modules for an ordinary Node.js program. First, write a Lingua Franca program (`Foo.lf`) and compile it. It may not type check if if you're [importing modules in the preamble](#) and you haven't installed the modules yet, but compiling your program will cause the TypeScript code generator to [produce a project](#) for your program. There should now be a package.json file in the same directory as your .lf file. Open a terminal and navigate to that directory. You can use the standard [`npm install`](#) command to install modules for your TypeScript reactors.

The important takeaway here is with the package.json file and the compiled JavaScript in the Foo/dist/ directory, you have a standard Node.js program that executes as such. You can modify and debug it just as you would a Node.js program.

Distributed Execution

NOTE: Distributed execution of Lingua Franca programs is at an early stage of development with many missing capabilities and a rather brittle execution. It is ready for experimentation, but not yet for deployment of serious systems. The capability has been tested on macOS and Linux, and there are no plans currently to support Windows systems.

A distributed Lingua Franca program is called a **federation**. Each reactor within the main reactor is called a **federate**. The LF compiler generates a separate program for each federate and synthesizes the code for the federates to communicate. The federates can be distributed across networks and eventually will be able to be written in different target languages, although this is not yet supported.

In addition to the federates, there is a program called the **RTI**, for **runtime infrastructure**, that coordinates startup and shutdown and may, if the coordination is centralized, mediate communication. The RTI needs to be compiled and installed separately on the system before any federation can execute.

It is possible to encapsulate federates in Docker containers for deployment. See containerized execution.

Installation of the RTI

Federated execution requires installation of a separate stand-alone program called the Runtime Infrastructure or **RTI**. At the current time, the only way to install this is from source files:

```
git clone https://github.com/lf-lang/reactor-c.git
cd reactor-c/core/federated/RTI/
mkdir build && cd build
cmake ../
make
sudo make install
```

The above will create a program called `RTI` and install it at `/usr/local/bin/RTI`. Once this program is available in your path, you can compile and execute federated Lingua Franca programs using Epoch, VS Code, or the command-line tools. For more details, see the README file.

Minimal Example

A minimal federated execution is specified by using the **federated** keyword instead of **main** for the main federate. An example is given below:

```
target TypeScript {
    // FIXME: This should work with timeout: 0 msec.
    timeout: 1 msec
}

reactor Source {
    output out:string;
    reaction(startup) -> out {=
        out = "Hello World!";
    =}
}
reactor Destination {
    input inp:string;
    reaction(inp) {=
        console.log("Received: " + inp);
    =}
}

federated reactor Federated {
    s = new Source();
    d = new Destination();
    s.out -> d.inp;
}
```

The **federated** keyword tells the code generator that the program is to be split into several distinct programs, one for each top level reactor.

When you run the code generator on `src/Federated.lf` containing the above code, the following three programs will appear:

```
bin/Federated
src-gen/dist/Federated/Federated_s.js
src-gen/dist/Federated/Federated_d.js
```

The root name, `Federated`, is the name of the .lf file from which these are generated (and the name of the main reactor, which is required to match if it is specified). The suffixes "_s" and "_d" come from the names of the top-level instances. There will always be one federate for each top-level reactor instance.

To run the program, you can simply run `bin/Federated`, which is a `bash` script that launches the RTI and two other programs, `Federated_s` and `Federated_d`. Alternatively, you can manually execute the RTI followed by the two federate programs by starting them on the command line. It is best to use three separate terminal windows (so that outputs from the three programs do not get jumbled together) to execute the following commands:

```
RTI -n 2
node src-gen/Federated/dist/Federated_s.js
node src-gen/Federated/dist/Federated_d.js
```

The `-n` argument to the `RTI` specifies that there it should expect two federates to join the federation.

Upon running the program, you will see information printed about the starting and ending of the federation, and buried in the output will be this line:

```
Received: Hello World!
```

Federation ID

You may have several federations running on the same machine(s) or even several instances of the same federation. In this case, it is necessary to distinguish between the federations. To accomplish this, you can pass a `-i` or `--id` parameter to the RTI and its federates with an identifier that is unique to the particular federation. For example,

```
RTI -n 2 -i myFederation
node src-gen/Federated/dist/Federated_s.js -i myFederation
node src-gen/Federated/dist/Federated_d.js -i myFederation
```

Each federate must have the same ID as the RTI in order to join the federation. The `bash` script that executes each of the components of the federation automatically generates a unique federation ID each time you run it.

Coordinated Start

When the above programs execute, each federate registers with the RTI. When all expected federates have registered, the RTI broadcasts to the federates the logical time at which they should start execution. Hence, all federates start at the same logical time.

The starting logical time is determined as follows. When each federate starts executing, it sends its current physical time (drawn from its real-time clock) to the RTI. When the RTI has heard from all the federates, it chooses the largest of these physical times, adds a fixed offset (currently one second), and broadcasts the resulting time to each federate.

When a federate receives the starting time from the RTI, if it is running in realtime mode (the default), then it will wait until its local physical clock matches or exceeds that starting time. Thus, to the extent that the machines have [synchronized clocks](#), the federates will all start executing at roughly the same physical time, a physical time close to the starting logical time.

Coordinated Shutdown

Coordinating the shutdown of a distributed program is discussed in [Termination](#).

Communication Between Federates

When one federate sends data to another, by default, the timestamp at the receiver will match the timestamp at the sender. You can also specify a logical delay on the communication using the **after** keyword. For example, if we had instead specified

```
s.out -> p.in after 200 msec;
```

then the timestamp at the receiving end will be incremented by 200 msec compared to the timestamp at the sender.

The preservation of timestamps across federates implies some constraints (unless you use [physical connections](#)). How these constraints are managed depends on whether you choose **centralized** or **decentralized** coordination.

Centralized Coordination

In the **centralized** mode of coordination (the default), the RTI regulates the advancement of time in each of the federates in order to ensure that the logical time semantics of Lingua Franca is respected. If the `p` federate above has an event with timestamp *t* that it wants to react to (it is the earliest event in its event queue), then it needs to get the OK from the RTI to advance its logical time to *t*. The RTI grants this time advance only when it can assure that `p` has received all messages that it will ever receive with timestamps *t* or less.

First, note that, by default, logical time on each federate never advances ahead of physical time, as reported by its local physical clock. Consider the consequences for the above connection. Suppose the timestamp of the message sent by `s` is *t*. This message cannot be sent before the local clock at `s` reaches *t* and also cannot be sent before the RTI grants to `s` a time advance to *t*. Since `s` has no federates upstream of it, the RTI will always grant it such a time advance (in fact, it does not even wait for a response from the RTI).

Suppose that the communication latency is L. That is, it takes L time units (in physical time) for a message to traverse the network. Then the `p` federate will not see the message from `s` before physical time $t + L$, where this physical time is measured by the physical clock on `s` 's host. If that clock differs from the clock on `p` 's host by E, then `p` will see the message at physical time $t + E + L$, as measured by its own clock. Let the value of the **after** specification (200 msec above) be a. Then the timestamp of the received message is $t + a$. The relationship between logical and physical times at the receiving end (the `p` federate), therefore, will depend on the relationship between a and $E + L$. If, for example, $E + L > a$, then federate `p` will lag behind physical time by at least $E + L - a$.

Assume the RTI has granted a time advance to t to federate `s` . Hence, `s` is able to send a message with timestamp t. The RTI now cannot grant any time advance to `p` that is greater than or equal to $t + a$ until the message has been delivered to `p` . In centralized coordination, all messages flow through the RTI, so the RTI will deliver a **Tag Advance Grant** (**TAG**) message to `p` only after it has delivered the message.

If $a > E + L$, then the existence of this communication does not cause `p` 's logical time to lag behind physical time. This means that if we were to modify `p` to have a **physical action**, the RTI will be able to immediately grant a **TAG** to `p` to advance the timestamp of that physical action. However, if $a < E + L$, then the RTI will delay granting a time advance to `p` by at least $E + L - a$. Hence, $E + L - a$ represents an additional latency in the processing of physical actions! This latency could present a problem for meeting deadlines. For this reason, if there are physical actions or deadlines at a federate that receives network messages, it is desirable to have **after** delays on the connection to that federate larger than any expected $E + L$. This way, there is no additional latency to processing physical actions at this federate and no additional risk of missing deadlines.

If, in addition, the physical clocks on the hosts are allowed to drift with respect to one another, then E can grow without bound, and hence the lag between logical time and physical time in processing events can grow without bound. This is mitigated either by hosts that themselves realize some clock synchronization algorithm, such as NTP or PTP, or by utilizing Lingua Franca's own built in clock synchronization. If the federates lack physical actions and deadlines, however, then unsynchronized clocks present no semantic problem if you are using centralized coordination. However, because of logical time chases physical time, federates will slow to match the slowest clock of federates upstream of them.

With centralized coordination, all messages (except those on physical connections) go through the RTI. This can create a bottleneck and a single point of failure. To avoid this bottleneck, you can use decentralized coordination.

Decentralized Coordination

The default coordination between mechanisms is **centralized**, equivalent to specifying the target property:

```
coordination: centralized
```

An alternative is **decentralized** coordination, which extends a technique realized in [PTIDES](#) and [Google Spanner](#), a globally distributed database system:

```
coordination: decentralized
```

With decentralized coordination, the RTI coordinates startup, shutdown, and clock synchronization, but is otherwise not involved in the execution of the distributed program.

In decentralized coordination, each federate and some reactions have a **safe-to-process** (**STP**) offset. When one federate communicates with another, it does so directly through a dedicated socket without going through the RTI. Moreover, it does not consult the RTI to advance logical time. Instead, it can advance its logical time to t when its physical clock matches or exceeds $t +$ STP.

By default, the STP is zero. An STP of zero is OK for any federate where either *every* logical connection into the federate has a sufficiently large **after** clause, or the federate has only one upstream federate sending it messages and it has no local timers or actions. The value of the **after** delay on each connection must exceed the sum of the [clock synchronization](#) error E, a bound L on the network latency, and the time lag on the sender D (the physical time at which it sends the message minus the timestamp of the message). The sender's time lag D can be enforced by using a **deadline**. For example:

```
WARNING: No source file found: ../code/ts/src/DecentralizedTimerAfter.lf
```

This example inherits from the Federated example above. In this example, as long as the messages from federate `c` arrive at federate `p` within 10 msec, all messages will be processed in tag order, as with an unfederated program.

An alternative to the **after** delays is to add an STP offset to downstream federates, as in the following example:

```
WARNING: No source file found: ../code/ts/src/DecentralizedTimerSTP.lf
```

Here, a parameter named `STP_offset` (not case sensitive) gives a time value, and the federate waits this specified amount of time (physical time) beyond a logical time t before advancing its logical time to t. In the above example, reactions to the timer events will be delayed by the amount specified by the `STP_offset` parameter. Just as with the use of **after**, if the `STP_offset` exceeds the sum of network latency, clock synchronization error, and execution times, then all events will be processed in tag order.

Of course, the assumptions about network latency, etc., can be violated in practice. Analogous to a deadline violation, Lingua Franca provides a mechanism for handling such a violation by providing an STP violation handler. The pattern is:

If the tag at which this reaction is to be invoked (the value returned by `lf_tag()`) exceeds the tag of an incoming message `in` (the current tag has already advanced beyond the intended tag of `in`), then the `STP` violation handler will be invoked instead of the normal reaction. Within the body of the STP handler, the code can access the intended tag of `in` using `in->intended_tag`, which has two fields, a timestamp `in->intended_tag.time` and a microstep `in->intended_tag.microstep`. The code can then ascertain the severity of the error and act accordingly. For example:

```
WARNING: No source file found: ../code/ts/src/DecentralizedTimerAfterHandle
```

For more advanced users, the LF API provides two functions that can be used to dynamically adjust the STP:

```
interval_t lf_get_stp_offset();
void lf_set_stp_offset(interval_t offset);
```

Using these functions, however, is a pretty advanced operation.

Physical Connections

Coordinating the execution of the federates so that timestamps are preserved is tricky. If your application does not require the deterministic execution that results from preserving the timestamps, then you can alternatively specify a **physical connection** as follows:

```
source.out ~> print.in;
```

The tilde specifies that the timestamp of the sender should be discarded. A new timestamp will be assigned at the receiving end based on the local physical clock, much like a **physical action**. To distinguish it from a physical connection, the normal connection is called a **logical connection**.

There are a number of subtleties with physical connections. One is that if you specify an `after` clause, for example like this:

```
count.out ~> print.in after 10 msec;
```

then what does this mean? At the receiving end, the timestamp assigned to the incoming event will be the current physical time plus 10 msec.

Prerequisites for Distributed Execution

In the above example, all of the generated programs expect to run on localhost. This is the default. With these defaults, every federate has to run on the same machine as the RTI because localhost is not a host that is visible from other machines on the network. In order to run federates or the RTI on remote machines, you can specify a domain name or IP address for the RTI and/or federates.

In order for a federated execution to work, there is some setup required on the machines to be used. First, each machine must be running on `ssh` server. On a Linux machine, this is typically done with a command like this:

```
sudo systemctl <start|enable> ssh.service
```

Enable means to always start the service at startup, whereas start means to just start it this once. On macOS, open System Preferences from the Apple menu and click on the "Sharing" preference panel. Select the checkbox next to "Remote Login" to enable it.

It will also be much more convenient if the launcher does not have to enter passwords to gain access to the remote machine. This can be accomplished by installing your public key (typically found in `~/.ssh/id_rsa.pub`) in `~/.ssh/authorized_keys` on the remote host.

Second, the RTI must be installed on the remote machine. See [instructions for installation the RTI](#).

Specifying RTI Hosts

You can specify a domain name on which the RTI should run as follows:

```
federated reactor DistributedCount at www.example.com {
    ...
}
```

You can alternatively specify an IP address (either IPv4 or IPv6):

```
federated reactor DistributedCount at 10.0.0.198 { ... }
```

By default, the RTI starts a socket server on port 15045, if that port is available, and increments the port number by 1 until it finds an available port. The number of increments is limited by a target-specific number. In the C target, in rti.h, STARTING_PORT defines the number 15045 and PORT_RANGE_LIMIT limits the range of ports attempted (currently 1024).

You can also specify a port for the RTI to use as follows:

```
federated reactor DistributedCount at 10.0.0.198:8080 { ... }
```

If you specify a specific port, then it will use that port if it is available and fail otherwise. The above changes this to port 8080.

Note that if the machine uses DHCP to obtain its address, then the generated code may not work in the future since the address of the machine may change in the future.

Address 0.0.0.0: The default host, `localhost` is used if no address is specified. Using `localhost` requires that the generated federates run on the local machine. This is ideal for testing. If you use `0.0.0.0`, then you are also specifying that the local machine (the one performing the code generation) will be the host, but now the process(es) running on this local machine can establish connections with processes on remote machines. The code generator will determine the IP address of the local machine, and any other hosts that need to communicate with reactors on the local host will use the current IP address of that local host at the time of code generation.

Specifying Federate Hosts

A federate may be mapped to a particular remote machine using a syntax like this:

```
count = new Count() at user@host:port/path;
```

The `port` is ignored in **centralized** mode because all communication is routed through the RTI, but in **decentralized** mode it will specify the port on which a socket server listens for incoming connections from other federates.

If any federate has such a remote designator, then a `Federation_distribute.sh` shell script will be generated. This script will distribute the generated code for the RTI to the remote machine at the specified directory.

You can also specify a user name on the remote machine for cases where the username will not match whoever launches the federation:

```
federated reactor DistributedCount at user@10.0.0.198:8080 { ... }
```

The general form of the host designation is

```
federated reactor DistributedCount at user@host:port/path { ... }
```

where `user@`, `:port`, and `/path` are all optional. The `path` specifies the directory on the remote machine (relative to the home directory of the user) where the generated code will be put. The `host` should be an IPv4 address (e.g. `93.184.216.34`), IPv6 address (e.g. `2606:2800:220:1:248:1893:25c8:1946`), or a domain name (e.g. `www.example.com`). It can

also be `localhost` or `0.0.0.0`. The host can be remote as long as it is accessible from the machine where the programs will be started.

If `user@` is not given, then it is assumed that the username on the remote host is the same as on the machine that launches the programs. If `:port` is not given, then it defaults to port 15045. If `/path` is not given, then `~user/LinguaFrancaRemote` will be the root directory on the remote machine.

Clock Synchronization

Both centralized and decentralized coordination have some reliance on clock synchronization. First, the RTI determines the start time of all federates, and the actually physical start time will differ by the extent that their physical clocks differ. This is particularly problematic if clocks differ by hours or more, which is certainly possible. If the hosts on which you are running run a clock synchronization algorithm, such as NTP or PTP, then you may not need to be concerned about this at all. Windows, Mac, and most versions of Linux, by default, run NTP, which synchronizes their clocks to some remote host. NTP is not particularly precise, however, so clock synchronization error can be hundreds of milliseconds or larger. PTP protocols are much more precise, so if your hosts derive their physical clocks from a PTP implementation, then you probably don't need to do anything further. Unfortunately, as of this writing, even though almost all networking hardware provides support for PTP, few operating systems utilize it. We expect this to change when people have finally understood the value of precise clock synchronization.

If your host is not running any clock synchronization, or if it is running only NTP and your application needs tighter latencies, then Lingua Franca's own built-in clock synchronization may provide better precision, depending on your network conditions. Like NTP, it realizes a software-only protocol, which are much less precise than hardware-supported protocols such as PTP, but if your hosts are on the same local area network, then network conditions may be such that the performance of LF clock synchronization will be much better than NTP. If your network is equipped with PTP, you will want to disable the clock synchronization in Lingua Franca by specifying in your target properties the following:

```
clock-sync: off
```

When a federation is mapped onto multiple machines, then, by default, any federate mapped to a machine that is not the one running the RTI will attempt during startup to synchronize its clock with the one on the machine running the RTI. The determination of whether the federate is running on the same machine is determined by comparing the string that comes after the `at` clause between the federate and the RTI. If they differ at all, then they will be treated as if the federate is running on a different machine even if it is actually running on the same machine. This default behavior can be obtained by either specifying nothing in the target properties or saying:

```
clock-sync: initial
```

This results in clock synchronization being done during startup only. To account for the possibility of your clocks drifting during execution of the program, you can alternatively specify:

```
clock-sync: on
```

With this specification, in addition to synchronization during startup, synchronization will be redone periodically during program execution.

Clock Synchronization Options

A number of options can be specified using the `clock-sync-options` target parameter. For example:

```
clock-sync-options: {local-federates-on: true, test-offset: 200 msec}
```

The supported options are:

- `local-federates-on` : Should be `true` or `false` . By default, if a federate is mapped to the same host as the RTI (using the `at` keyword), then clock synchronization is turned off. This assumes that the federate will be using the same clock as the RTI, so there is no point in performing clock synchronization. However, sometimes it is useful to force clock synchronization to be run even in this case, for example to test the performance of clock synchronization. To force clock synchronization on in this case, set this option to `true` .

- `test-offset` : The value should be a time value with units, e.g. `200 msec` . This will establish an artificial fixed offset for each federate's clock of one plus the federate ID times the time value given. For example, with the value `200 msec` , a fixed offset of 200 milliseconds will be set on the clock for federate 0, 400 msec on the clock of federate 1, etc.

- `period` : A time value (with units) that specifies how often runtime clock synchronization will be performed if it is turned on. The default is `5 msec` .

- `attenuation` : A positive integer specifying a divisor applied to the estimated clock error during runtime clock synchronization when adjusting the clock offset. The default is `10` . Making this number bigger reduces each adjustment to the clock. Making the number equal to `1` means that each round of clock synchronization fully applies its estimated clock synchronization error.

- `trials` : The number of rounds of message exchange with the RTI in each clock synchronization round. This defaults to `10` .

Termination

Shutdown Reactions

There are several mechanisms for terminating a Lingua Franca in an orderly fashion. All of these mechanisms result in a **final tag** at which any reaction that declares **shutdown** as a trigger will be invoked (recall that a **tag** is a tuple (**logical time**, **microstep**)). Other reactions may also be invoked at this final tag, and the order in which reactions are invoked will be constrained by the normal precedence rules.

If a reaction triggered by **shutdown** produces outputs, then downstream reactors will also be invoked at the final tag. If the reaction schedules any actions by calling `schedule()`, those will be ignored. In fact, any event after the final tag will be ignored. After the completion of the final tag, the program will exit.

There are four ways to terminate a program:

- **Timeout**: The program specifies the last logical time at which reactions should be triggered.
- **Starvation**: At the conclusion of some tag, there are no events in the event queue at future tags.
- **Stop request**: Some reaction requests that the program terminate.
- **External signal**: Program is terminated externally using operating services like control-C or `kill`.

We address each of these in turn.

Timeout

The [target property `timeout`](#) specifies the last logical time at which reactions should be triggered. The last invocation of reactions will be at tag (`timeout`, 0).

There is a significant subtlety when using [physical connections](#), which are connections using the syntax `~>`. Such connections specify that the tag at the receiving end will be based on the physical time at which the message is received. If the tag assigned at the receiving end is greater than the final tag, then the message is lost. Hence, **messages sent near the `timeout` time are likely to be lost!**

Starvation

If a Lingua Franca program has no [physical actions](), and if at any time during execution there are no future events waiting to be processed, then there is no possibility for any more reactions to occur and the program will exit. This situation is called **starvation**. If there is a **timer** anywhere in the program with a period, then this condition never occurs.

One subtlety is that reactions triggered by **shutdown** will be invoked one microstep later than the last tag at which there was an event. They cannot be invoked at the same tag because it is only after that last tag has completed that the runtime system can be sure that there are no future events. It would not be correct to trigger the **shutdown** reactions at that point because it would be impossible to respect the required reaction ordering.

Starvation termination is not currently implemented for federated execution. You will need to use one of the other mechanisms to terminate a federated program.

Stop Request

If a reaction calls the built-in `request_stop()` function, then it is requesting that the program cease execution as soon as possible. This cessation will normally occur in the next microstep. The current tag will be completed as normal. Then the tag will be advanced by one microstep, and reactions triggered by **shutdown** will be executed, along with any other reactions with triggers at that tag, with all reactions executed in precedence order.

In a federated execution, things are more complicated. In general, it is not possible to cease execution in the next microstep because this would mean that every federate has a communication channel to every other with delay equal to one microstep. This does not create a causality loop, but it means that all federates have to advance time in lockstep, which creates a global barrier synchronization that would likely kill performance. It would also make decentralized coordination impossible because the safe-to-process (STP) threshold for all federates would diverge to infinity.

For **centralized coordination**, when a reaction in a federate calls `request_stop()`, the federate sends a **STOP_REQUEST** message to the RTI with its current timestamp t as a payload and completes execution of any other reactions triggered at the current tag. It then blocks, waiting for a **STOP_GRANTED** message with a timestamp payload s. If $s > t$, then it sets `timeout` = s and continues executing, using the timeout mechanism (see above) to stop. If $s = t$, then schedules the shutdown phase to occur one microstep later, as in the unfederated case.

When the RTI receives a **STOP_REQUEST** message from a federate, it forwards it to all other federates and waits for a reply from all. Each reply will have a timestamp payload. The RTI chooses s, the largest of these timestamps, and sends a **STOP_GRANTED** message to all federates with payload s.

When a federate receives a **STOP_REQUEST** message, it replies with its current logical time t, completes its current tag (if one is progress), and blocks, waiting for a **STOP_GRANTED** message from the RTI. When it gets the reply with payload s, if $s > t$, then it sets `timeout` $= s$ and continues executing, using the timeout mechanism (see above) to stop. If $s = t$, then it schedules the shutdown phase to occur one microstep later, as in the unfederated case.

External Signal

A control-C or other kill signal to a running Lingua Franca program will cause execution to stop immediately.

For federated programs, each federate and the RTI catches external signals to shut down in an orderly way.

When a federate gets such an external signal (e.g. control-C), it sends a **RESIGN** message to the RTI and an **EOF** (end of file) on each socket connection to another federate. It then closes all sockets and shuts down. The RTI and all other federates should continue running until some other termination condition occurs.

When the RTI gets such an external signal (e.g. control-C), it broadcasts a **STOP_REQUEST** message to all federates, waits for their replies (with a timeout in case the federate or the network has failed), chooses the maximum timestamp s on the replies, broadcasts a **STOP_GRANTED** message to all federates with payload s, and waits for **LOGICAL_TIME_COMPLETE** messages as above.