This copy of the Lingua Franca handbook was created on Sunday, April 3, 2022 against commit `d32d0b`.

Table of Contents

A First Reactor

This page is showing examples in the target language C C++ Python TypeScript Rust. You can change the target language in left sidebar. See setup for C. See setup for C++. See setup for Python. See setup for TypeScript. See setup for Rust.

Minimal Example

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

```
target C;
main reactor {
    reaction(startup) {=
        printf("Hello World.\n");
    =}
}
```

```
target Cpp;
main reactor {
    reaction(startup) {=
        cout << "Hello World.\n";
    =}
}
```

```
target Python;
main reactor {
    reaction(startup) {=
        print("Hello World.")
    =}
}
```

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

```
target Rust;
main reactor {
    reaction(startup) {=
        println!("Hello World.");
    =}
}
```

Every Lingua Franca program begins with a target specification 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, delimitted by `{= ... =}`, is ordinary CCppPythonTypeScriptRust code which, as we will see, has access to a number of functions and variables specific to Lingua Franca.

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/Minimal.lf
... output from the code generator and compiler ...
```

After this completes, two additional directories will have been created within the projet root, `bin` and `src-gen`. The `bin` directory has an executable file called `HelloWorld`. Executing that file will result, not surprisingly, in printing "Hello World". The generated source files will be within the directory `src-gen`.

After this completes, an additional `src-gen` directory will have been created within the projet 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/Minimal.js

#####################################

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

    python3 ...path-to-project.../src-gen/Minimal/Minimal.py

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

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>]
}
```

```
[main] 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>]
    [const] method <name>(<parameters>):<type> {= ... body ...=}
    reaction(<triggers>) [<uses>] [=> <effects>] {= ... body ...=}
    <instance-name> = new <class-name>([<parameter-assignments>])
    <instance-name> [, ...] => <instance-name> [, ...] [after <delay>]
}
```

```
[main or federated] reactor <class-name> [(<parameters>)] {
    input <name>
    output <name>
    state <name>(<value>)
    timer <name>([<offset>, [<period>]])
    logical action <name>
    physical action <name>
    [const] method <name>(<parameters>) {= ... body ...=}
    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.
'''
   Multi-line Python-style comment.
'''
```

Inputs and Outputs

This page is showing examples in the target language C C++ Python TypeScript Rust. You can change the target language in left sidebar.

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

Input and Output Declarations

Input and output declarations have the form:

```
input <name>:<type>
output <name>:<type>

input <name>
output <name>
```

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

```
target C;
reactor Double {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        SET(y, x->value * 2);
    =}
}

target Cpp;
reactor Double {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        y.set(x.value * 2);
    =}
}

WARNING: No source file found: ../code/py/src/Double.lf


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

```
WARNING: No source file found: ../code/rs/src/Double.lf
```

Notice how the input value is accessed and how the output value is set. This is done differently for each target language. See C Reactors C++ Reactors Python Reactors TypeScriupt Reactors Rust Reactors for detailed documentation of these mechanisms.

The **type** of a port is a type in the target language plus the special type **time**. A type may also be specified using a **code block**, delimited by the same delimeters `{= ... =}` that separate target language code from Lingua Franca code in reactions. Any valid target-language type designator can be given within these delimiters. See Lingua Franca Types for details.

Triggers, Effects, and Uses

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 C;
reactor Destination {
    input x:int;
    input y:int;
    reaction(x, y) {=
        int sum = 0;
        if (x->is_present) {
            sum += x->value;
        }
        if (y->is_present) {
            sum += y->value;
        }
        printf("Received %d.\n", sum);
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Destination.lf
```

```
WARNING: No source file found: ../code/py/src/Destination.lf
```

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

```
WARNING: No source file found: ../code/rs/src/Destination.lf
```

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 C target, the value read will be the most recently seen input value, or, if no input event has occurred at an earlier logical time, then zero or NULL, depending on the datatype of the input. FIXME. FIXME. In the TS target, the value will be **undefined**, a legitimate value in TypeScript. FIXME.

Reactions

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.

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.

Occassionally, 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-spefic reference documentation has more details about how this works.

Parameters and State Variables

This page is showing examples in the target language C C++ Python TypeScript Rust. You can change the target language in left sidebar.

Parameter Declaration

A reactor class definition can parameterized as follows:

```
reactor <class-name>(<param-name>:<type>(<expr>), ...) {
    ...
}
```

Each parameter has a *type annotation*, written `:<type>`, where `<type>` has one of the following forms:

- An identifier, such as `int`, possibly followed by a type argument, e.g. `vector<int>`.
- An array type `type[]` and `type[integer]`.
- The keyword **time**, which designates a time value.
- A code block delimitted by `{= ... =}`, where the contents is any valid type in the target language.
- A pointer type, such as `int*`.

Types ending with a `*` are treated specially by the C target. See Sending and Receiving Arrays and Structs in the C target documentation.

To use strings conveniently in the C target, the "type" `string` is an alias for `{=const char*=}`.

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

```
reactor <class-name>(<param-name>(<expr>), ... ) {
    ...
}
```

Each parameter must have a *default value*, written `(<expr>)`. An expression may be a numeric contant, 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 C;
reactor Scale(factor:int(2)) {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        SET(y, x->value * self->factor);
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Scale.lf

WARNING: No source file found: ../code/py/src/Scale.lf

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

WARNING: No source file found: ../code/rs/src/Scale.lf
```

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. In the C target, a `self` struct is provided that contains the parameter values.

State Declaration

A reactor declares a state variable as follows:

```
    state <name>:<type>(<value>);
```

The type can any of the same forms as for a parameter.

```
    state <name>(<value>);
```

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 C;
reactor Count {
    state count:int(0);
    output y:int;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        SET(y, self->count++);
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Count.lf

WARNING: No source file found: ../code/py/src/Count.lf

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

WARNING: No source file found: ../code/rs/src/Count.lf
```

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 trigger by a **timer**, discussed in the next section.

Time and Timers

This page is showing examples in the target language C C++ Python TypeScript Rust. You can change the target language in left sidebar.

Timers

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.

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 C;
main reactor Timer {
    timer t(0, 1 sec);
    reaction(t) {=
        printf("Logical time is %lld.\n", get_logical_time());
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Timer.lf
```

```
WARNING: No source file found: ../code/py/src/Timer.lf
```

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

```
WARNING: No source file found: ../code/rs/src/Timer.lf
```

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, `get_logical_time()` FIXME FIXME FIXME FIXME. 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 C;
main reactor TimeElapsed {
    timer t(0, 1 sec);
    reaction(t) {=
        printf(
            "Elapsed logical time is %lld.\n",
            get_elapsed_logical_time()
        );
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/TimeElapsed.lf
```

```
WARNING: No source file found: ../code/py/src/TimeElapsed.lf
```

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

```
WARNING: No source file found: ../code/rs/src/TimeElapsed.lf
```

See the C reactors documentation C++ reactors documentation Python reactors documentation TypeScript reactors documentation Rust reactors documentation 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 C;
main reactor TimeLag {
    timer t(0, 1 sec);
    reaction(t) {=
        interval_t t = get_elapsed_logical_time();
        interval_t T = get_elapsed_physical_time();
        printf(
            "Elapsed logical time: %lld, physical time: %lld, lag: %lld\n",
            t, T, T-t
        );
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/TimeLag.lf
```

```
WARNING: No source file found: ../code/py/src/TimeLag.lf
```

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

```
WARNING: No source file found: ../code/rs/src/TimeLag.lf
```

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](ocs/handbook/target-specification#command-line-arguments). For example, the following` timeout` property will cause the above timer with a period of one second to terminate after 11 events:

```
target C {
    timeout: 10 sec
}

target Cpp {
    timeout: 10 sec
}

target Python {
    timeout: 10 sec
}

target TypeScript {
    timeout: 10 sec
}

target Rust {
    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 C;
reactor TestCount(start:int(0), stride:int(1), num_inputs:int(1)) {
    state count:int(start);
    state inputs_received:int(0);
    input x:int;
    reaction(x) {=
        printf("Received %d.\n", x->value);
        if (x->value != self->count) {
            printf("ERROR: Expected %d.\n", self->count);
            exit(1);
        }
        self->count += self->stride;
        self->inputs_received++;
    =}
    reaction(shutdown) {=
        printf("Shutdown invoked.\n");
        if (self->inputs_received != self->num_inputs) {
            printf("ERROR: Expected to receive %d inputs, but got %d.\n",
                self->num_inputs,
                self->inputs_received
            );
            exit(2);
        }
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/TestCount.lf

WARNING: No source file found: ../code/py/src/TestCount.lf

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

WARNING: No source file found: ../code/rs/src/TestCount.lf
```

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

This page is showing examples in the target language C C++ Python TypeScript Rust. You can change the target language in left sidebar.

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 C {
    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, num_inputs = 11);
    c.y -> s.x;
    s.y -> t.x;
}
```

```
WARNING: No source file found: ../code/cpp/src/RegressionTest.lf

WARNING: No source file found: ../code/py/src/RegressionTest.lf

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

WARNING: No source file found: ../code/rs/src/RegressionTest.lf
```

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:

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 contant, 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:

```
reactor A {
    output y:int
}
reactor B {
    input x:int
}
main reactor {
    a = new A()
    b1 = new B()
    b2 = new B()
    a.y -> b1.x
    a.y -> b2.x
}
```

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 <reactor class> as <alias2> from "<path>"
```

where `<reactor class>` 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 C;
import Count from "Count.lf";
import Scale from "Scale.lf";
import TestCount from "TestCount.lf";

reactor Container(stride:int(2)) {
    output y:int;
    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, num_inputs = 11);
    c.y -> t.x;
}
```

```
WARNING: No source file found: ../code/cpp/src/Hierarchy.lf

WARNING: No source file found: ../code/py/src/Hierarchy.lf

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

WARNING: No source file found: ../code/rs/src/Hierarchy.lf
```

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:

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 C {
    timeout: 3 secs
}
main reactor Alignment {
    state s:int(0);
    timer t1(100 msec, 100 msec);
    timer t2(200 msec, 200 msec);
    timer t4(400 msec, 400 msec);
    reaction(t1) {=
        self->s += 1;
    =}
    reaction(t2) {=
        self->s -= 2;
    =}
    reaction(t4) {=
        printf("s = %d\n", self->s);
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Alignment.lf

WARNING: No source file found: ../code/py/src/Alignment.lf

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

WARNING: No source file found: ../code/rs/src/Alignment.lf
```

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 C;
reactor Overwriting {
    output y:int;
    state s:int(0);
    timer t1(100 msec, 100 msec);
    timer t2(200 msec, 200 msec);
    reaction(t1) -> y {=
        self->s += 1;
        SET(y, self->s);
    =}
    reaction(t2) -> y {=
        self->s -= 2;
        SET(y, self->s);
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Overwriting.lf

WARNING: No source file found: ../code/py/src/Overwriting.lf

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

WARNING: No source file found: ../code/rs/src/Overwriting.lf
```

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 C;
import Overwriting from "Overwriting.lf";
main reactor {
    s = new Overwriting();
    reaction(s.y) {=
        if (s.y->value != 0 && s.y->value != 1) {
            error_print_and_exit("Outputs should only be 0 or 1!");
        }
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Contained.lf

WARNING: No source file found: ../code/py/src/Contained.lf

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

WARNING: No source file found: ../code/rs/src/Contained.lf
```

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

Method Declaration

The CCppPythonTypeScriptRust target does not currently support methods.

A method declaration has one of the forms:

```
method <name>();
method <name>():<type>;
method <name>(<argument_name>:<type>, ...);
method <name>(<argument_name>:<type>, ...):<type>;
```

The first form defines a method with no arguments and no return value. The second form defines a method with the return type `<type>` but no arguments. The third form defines a method with a comma-separated list of arguments given by their name and type, but without a return value. Finally, the fourth form is similar to the third, but adds a return type.

The **method** keywork can optionally be prefixed with the **const** qualifier, which indicates that the method is read only.

Methods are particularly useful in reactors that need to perform certain operations on state variables and/or parameters that are shared between reactions or that are too complex to be implemented in a single reaction. Analogous to class methods, methods in LF can access all state variables and parameters, and can be invoked from all reaction bodies or from other methods. Consider the following example:

```
WARNING: No source file found: ../code/c/src/Methods.lf
```

```
target Cpp;
main reactor Methods {
    state foo:int(2);
    const method getFoo(): int {=
        return foo;
    =}
    method add(x:int) {=
        foo += x;
    =}
    reaction(startup){=
        std::cout << "Foo is initialized to " << getFoo() << '\n';
        add(40);
        std::cout << "2 + 40 = " << getFoo() << '\n';
    =}
}
```

```
WARNING: No source file found: ../code/py/src/Methods.lf
```

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

```
WARNING: No source file found: ../code/rs/src/Methods.lf
```

This reactor defines two methods `getFoo` and `add`. `getFoo` is qualified as a const method, which indicates that it has read-only access to the state variables. This is direclty translated to a C++ const method in the code generation process. The `getFoo` method receives no arguments and returns an integer (`int`) indicating the current value of the `foo` state variable. The `add` method returns nothing (`void`) and receives one interger argument, which it uses to increment `foo`. Both methods are visible in all reactions of the reactor. In this example, the reaction to startup calls both methods in order to read and modify its state.

Extending Reactors

This page is showing examples in the target language C C++ Python TypeScript Rust. You can change the target language in left sidebar.

Extending a Base Reactor

Lingua Franca supports defining a reactor class as an extension (or subclass), as in the following example:

```
target C;
reactor A {
    input a:int;
    output out:int;
    reaction(a) -> out {=
        SET(out, a->value);
    =}
}
reactor B extends A {
    input b:int;
    reaction(a, b) -> out {=
        SET(out, a->value + b->value);
    =}
}
```

```
WARNING: No source file found: ../code/cpp/src/Extends.lf

WARNING: No source file found: ../code/py/src/Extends.lf

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

WARNING: No source file found: ../code/rs/src/Extends.lf
```

Here, the base class `A` has a single output that it writes to in reaction to an input. The subclass inherits the input, the output, and the reaction of `A`, and adds its own input `b` and reaction. When an input event `a` arrives, both reactions will be invoked, but, once again, in a well-defined order. The reactions of the base class are invoked before those of the derived class. So in this case, `B` will overwrite the output produced by `A`.

One limitation is that a subclass cannot have ports, actions, or state variables with the same names as those in the base class. The names must be unique.

A subclass can extend more than one base class by just providing a comma-separated list of base classes. If reactions in multiple base classes are triggered at the same tag, they will be invoked in the same order that they appear in the comma-separated list.

Actions

Action Declaration

An **action**, like an input, can cause reactions to be invoked. Whereas inputs are provided by other reactors, actions are scheduled by this reactor itself, either in response to some observed external event or as a delayed response to some input event. The action can be scheduled by a reactor by invoking a **schedule** function in a reaction or in an asynchronous callback function.

An action declaration is either physical or logical:

```
physical action name(min_delay, min_spacing, policy):type;
> logical action name(min_delay, min_spacing, policy):type;
```

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*, etc. The *min_delay* and *min_spacing* have to be a time value. The *policy* argument is a string that can be one of the following: `'defer'` (default), `'drop'`, or `'replace'`.

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 above, and whether the action is physical or logical.

If the **logical** keyword is given, then the tag assigned to the event resulting from a call to **schedule** function is computed as follows. First, let *t* be the *current logical time*. For a logical action, the `schedule` function must be invoked from within a reaction (synchronously), so *t* is just the logical time of that reaction.

The (preliminary) tag of the action is then just *t* plus *min_delay* plus the *offset* argument to **schedule** function.

If the **physical** keyword is given, then the physical clock on the local platform is used as the timestamp assigned to the action. Moreover, for a physical action, unlike a logical action, the `schedule` function can be invoked from outside of any reaction (asynchronously), e.g. from an interrupt service routine or callback function.

If a *min_spacing* has been declared, then a minimum distance between the tags of two subsequently scheduled events on the same action is enforced. If the preliminary tag is closer to the tag of the previously scheduled event (if there is one), then *policy* determines how the given constraints is enforced.

- `'drop'`: the new event is dropped and `schedule` returns without having modified the event queue.
- `'replace'`: the payload of the new event is assigned to the preceding event if it is still pending in the event queue; no new event is added to the event queue in this case. If the preceding event has already been pulled from the event queue, the default `'defer'` policy is applied.
- `'defer'`: the event is added to the event queue with a tag that is equal to earliest time that satisfies the minimal spacing requirement. Assuming the tag of the preceding event is *t_prev*, then the tag of the new event simply becomes *t_prev* + *min_spacing*.

Note that while the `'defer'` policy is conservative in the sense that it does not discard events, it could potentially cause an unbounded growth of the event queue.

In all cases, the logical time of a new event will always be strictly greater than the logical time at which it is scheduled by at least one microstep (see the Time section).

The default *min_delay* is zero. The default *min_spacing* is undefined (meaning that no minimum spacing constraint is enforced). If a `min_spacing` is defined, it has to be strictly greater than zero, and greater than or equal to the time precision of the target (for the C target, it is one nanosecond).

The *min_delay* parameter in the **action** declaration is static (set at compile time), while the *offset* parameter given to the schedule function may be dynamically set at runtime. Hence, for static analysis and scheduling, the **action**'s' *min_delay* parameter can be assumed to be a *minimum delay* for analysis purposes.

Superdense 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*. Two events are logically **simultaneous** only if *both* the logical time and the microstep are equal. But only the logical time is used to align behavior with physical time. For that purpose, the microstep is ignored.

Discussion

Logical actions are used to schedule events at a future logical time relative to the current logical time. Physical time is ignored. They must be scheduled within reactions, and the timestamp of the scheduled event will be relative to the current logical time of the reaction that schedules them. It is an error to schedule a logical action asynchronously, outside of the context of a reaction. Asynchronous actions are required to be **physical**.

Physical actions are typically used to assign timestamps to externally triggered events, such as the arrival of a network message or the acquisition of sensor data, where the time at which these

external events occurs is of interest. There are (at least) three interesting use cases:

1. An asynchronous event, such as a callback function or interrupt service routine (ISR), is invoked at a physical time *t* and schedules an action with timestamp *T*=*t*. To get this behavior, just set the physical action to have *min_delay* = 0 and call the schedule function with *offset* = 0. The *min_spacing* can be useful here to prevent these external events from overwhelming the software system.
2. A periodic task that is occasionally modified by a sporadic sensor. In this case, you can set *min_delay* = *period* and call schedule with *offset* = 0. The resulting timestamp of the sporadic sensor event will always align with the periodic events. This is similar to periodic polling, but without the overhead of polling the sensor when nothing interesting is happening.
3. You can impose a minimum physical time delay between an event's occurrence, such as a push of a button, and system response by adjusting the *offset*.

Actions With Values

If an action is declared with a *type*, then it can carry a **value**, a data value passed to the **schedule** function. This value will be available to any reaction that is triggered by the action. The specific mechanism, however, is target-language dependent. See the C target for an example.

Scheduling Future Reactions

Each target language provides some mechanism for scheduling future reactions. Typically, this takes the form of a `schedule` function that takes as an argument an action, a time interval, and (perhaps optionally), a payload. For example, in the C target, in the following program, each reaction to the timer `t` schedules another reaction to occur 100 msec later:

```
target C;
main reactor Schedule {
    timer t(0, 1 sec);
    logical action a;
    reaction(t) -> a {=
        schedule(a, MSEC(100));
    =}
    reaction(a) {=
        printf("Nanoseconds since start: %lld.\n", get_elapsed_logical_time
    =}
}
```

When executed, this will produce the following output:

```
Start execution at time Sun Aug 11 04:11:57 2019
plus 919310000 nanoseconds.
Nanoseconds since start: 100000000.
Nanoseconds since start: 1100000000.
Nanoseconds since start: 2100000000.
...
```

This action has no datatype and carries no value, but, as explained below, an action can carry a value.

Asynchronous Callbacks

In targets that support multitasking, the `schedule` function, which schedules future reactions, may be safely invoked on a **physical action** in code that is not part of a reaction. For example, in the multithreaded version of the C target, `schedule` may be invoked in an interrupt service routine. The reaction(s) that are scheduled are guaranteed to occur at a time that is strictly larger than the current logical time of any reactions that are being interrupted.

Superdense Time

Lingua Franca uses a concept known as **superdense time**, where two time values that appear to be the same are not logically simultaneous. At every logical time value, for example midnight on January 1, 1970, there exist a logical sequence of **microsteps** that are not simultaneous. The Microsteps example illustrates this:

```
target C;
reactor Destination {
    input x:int;
    input y:int;
    reaction(x, y) {=
        printf("Time since start: %lld.\n", get_elapsed_logical_time());
        if (x->is_present) {
            printf("  x is present.\n");
        }
        if (y->is_present) {
            printf("  y is present.\n");
        }
    =}
}
main reactor Microsteps {
    timer start;
    logical action repeat;
    d = new Destination();
    reaction(start) -> d.x, repeat {=
        SET(d.x, 1);
        schedule(repeat, 0);
    =}
    reaction(repeat) -> d.y {=
        SET(d.y, 1);
    =}
}
```

The `Destination` reactor has two inputs, `x` and `y`, and it simply reports at each logical time where either is present what is the logical time and which is present. The `Microsteps` reactor initializes things with a reaction to the one-time timer event `start` by sending data to the `x` input of `Destination`. It then schedules a `repeat` action.

Note that time delay in the call to `schedule` is zero. However, any reaction scheduled by `schedule` is required to occur **strictly later** than current logical time. In Lingua Franca, this is handled by scheduling the `repeat` reaction to occur one **microstep** later. The output printed, therefore, will look like this:

```
Time since start: 0.
  x is present.
Time since start: 0.
  y is present.
```

Note that the numerical time reported by `get_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.

Note that it is possible to write code that will prevent logical time from advancing except by microsteps. For example, we could replace the reaction to `repeat` in `Main` with this one:

```
    reaction(repeat) -> d.y, repeat {=
        SET(d.y, 1);
        schedule(repeat, 0);
    =}
```

This would create what is known as a **stuttering Zeno** condition, where logical time cannot advance. The output will be an unbounded sequence like this:

```
Time since start: 0.
  x is present.
Time since start: 0.
  y is present.
Time since start: 0.
  y is present.
Time since start: 0.
  y is present.
...
```

Startup and Shutdown Reactions

Two special triggers are supported, **startup** and **shutdown**. A reaction that specifies the **startup** trigger will be invoked at the start of execution of the model. The following two syntaxes have exactly the same effect:

```
    reaction(startup) {= ... =}
```

and

```
    timer t;
    reaction(t) {= ... =}
```

In other words, **startup** is a timer that triggers once at the first logical time of execution. As with any other reaction, the reaction can also be triggered by inputs and can produce outputs or schedule actions.

The **shutdown** trigger is slightly different. A shutdown reaction is specified as follows:

```
   reaction(shutdown) {= ... =}
```

This reaction will be invoked when the program terminates normally (there are no more events, some reaction has called a `request_stop()` utility provided in the target language, or the execution was specified to last a finite logical time). The reaction will be invoked at a logical time one microstep *later* than the last logical time of the execution. In other words, the presence of this reaction means that the program will execute one extra logical time cycle beyond what it would have otherwise, and that logical time is one microstep later than what would have otherwise been the last logical time.

If the reaction produces outputs, then downstream reactors will also be invoked at that later logical time. If the reaction schedules future reactions, those will be ignored. After the completion of this final logical time cycle, one microstep later than the normal termination, the program will exit.

Deadlines

Deadlines

Lingua Franca includes a notion of a **deadline**, which is a 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 timestamp 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. For example (see Deadline):

```
reactor Deadline() {
    input x:int;
    output d:int; // Produced if the deadline is violated.
    reaction(x) -> d {=
        printf("Normal reaction.\n");
    =} deadline(10 msec) {=
        printf("Deadline violation detected.\n");
        SET(d, x->value);
    =}
```

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.

The amount of the deadline, of course, can be given by a parameter.

A sometimes useful pattern is when a container reactor reacts to deadline violations in a contained reactor. The DeadlineHandledAbove example illustrates this:

```
target C;
reactor Deadline() {
    input x:int;
    output deadline_violation:bool;
    reaction(x) -> deadline_violation {=
        ... normal code to execute ...
    =} deadline(100 msec) {=
        printf("Deadline violation detected.\n");
        SET(deadline_violation, true);
    =}
}
main reactor DeadlineHandledAbove {
    d = new Deadline();
    ...
    reaction(d.deadline_violation) {=
        ... handle the deadline violation ...
    =}
}
```

Multiports and Banks

This page describes Lingua Franca language constructs support more scalable programs by providing a compact syntax for ports that can send or receive over multiple channels (called **multiports**) and multiple instances of a reactor class (called a **bank of reactors**). The examples given below include syntax of the C target for accessing the ports. Other targets use different syntax with target code, within the delimiters `{= ... =}`, but use the same syntax outside those delimiters.

Multiports

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

```
input[width] name:type; output[width] name:type;
```

where *width* is a positive integer. This can be given either as an integer literal or a parameter name. For targets that allow dynamic parametrization at runtime (like the C++ target), width can also be given by target code enclosed in `{=...=}`.

For example, (see MultiportToMultiport):

```
target C;
reactor Source {
    output[4] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
reactor Destination {
    input[4] in:int;
    reaction(in) {=
        int sum = 0;
        for (int i = 0; i < in_width; i++) {
            if (in[i]->is_present) sum += in[i]->value;
        }
        printf("Sum of received: %d.\n", sum);
    =}
}
main reactor MultiportToMultiport {
    a = new Source();
    b = new Destination();
    a.out -> b.in;
}
```

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. Running this program produces:

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

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 `if (in[i]->is_present) ...`. An event on any one of the channels is sufficient to trigger the reaction.

Sending and Receiving Via a Multiport

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. It is also possible to test whether a previous reaction has set an output value and to read what that value is. The exact syntax for this depends on the target language. In the C

target, the width is accessed with the variable `out_width`, and `out[i]` references the output channel to write to using the `SET` macro, as shown above. In addition, `out[i]->is_present` and `out[i]->value` are defined. For example, if we modify the above reaction as follows:

```
reactor Source {
    output[4] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            printf("Before SET, out[%d]->is_present has value %d\n", i, out
            SET(out[i], i);
            printf("AFTER set, out[%d]->is_present has value %d\n", i, out[
            printf("AFTER set, out[%d]->value has value %d\n", i, out[i]->v
        }
    =}
}
```

then we get the output:

```
Before SET, out[0]->is_present has value 0
AFTER set, out[0]->is_present has value 1
AFTER set, out[0]->value has value 0
Before SET, out[1]->is_present has value 0
AFTER set, out[1]->is_present has value 1
AFTER set, out[1]->value has value 1
Before SET, out[2]->is_present has value 0
AFTER set, out[2]->is_present has value 1
AFTER set, out[2]->value has value 2
Before SET, out[3]->is_present has value 0
AFTER set, out[3]->is_present has value 1
AFTER set, out[3]->value has value 3
Sum of received: 6.
```

If you access `out[i]->value` before any value has been set, the result is undefined.

In the Python target, multiports can be iterated on in a for loop (e.g., `for p in out`) or enumerated (e.g., `for i, p in enumerate(out)`) and the length of the multiport can be obtained by using the `len()` (e.g., `len(out)`) expression.

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 {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
```

In some targets such as the C++ target, parameters to the main reactor can be overwritten at the command line interface, allowing for dynamically scalable applications.

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 (see MultiportToMultiport2)

```
main reactor MultiportToMultiport2 {
    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 inputs 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 (see MultiportToBankMultiport):

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

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 BankToBankMultiport {
    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** with any type that can be assigned a non-negative integer value (in C, for example, `int`, `size_t`, or `uint32_t` will all work). If such a parameter is defined for the reactor, then when the reactor is instanced 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 (see BankToBank):

```
reactor Source(
    bank_index:int(0)
) {
    timer t(0, 200 msec);
    output out:int;
    state s:int(0);
    reaction(t) -> out {=
        SET(out, self->s);
        self->s += self->bank_index;
    =}
}
```

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

```
main reactor BankToBankMultiport(
    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 C;
reactor Child (
    bank_index:int(0)
) {
    reaction(startup) {=
        info_print("My bank index is %d.", self->bank_index);
        =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child();
}
main reactor {
    p = new[2] Parent();
}
```

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 is 0.
My bank index is 1.
My bank index is 0.
My bank index is 1.
```

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

```
target C;
reactor Child (
    bank_index:int(0),
    parent_bank_index:int(0)
) {
    reaction(startup) {=
        info_print(
            "My parent's bank index is %d.",
            self->parent_bank_index
            );
        =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
}
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 parent's bank index is 0.
My parent's bank index is 1.
My parent's bank index is 1.
My parent's bank index is 0.
```

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 C;
reactor Child (
    bank_index:int(0),
    parent_bank_index:int(0)
) {
    output out:int;
    reaction(startup) -> out {=
        SET(out, self->parent_bank_index + self->bank_index);
        =}
}
reactor Parent (
    bank_index:int(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
    reaction(c.out) {=
        for (int i=0; i < c_width; i++) {
            info_print("Received %d from child %d.", c[i].out->value, i);
        }
    =}
}
main reactor {
    p = new[2] Parent();
}
```

Note the usage of `c_width`, which holds the width of the `c` bank of reactors. Note that this syntax is target-specific. For example, in the Python target, `len(c)` can be used to get the width of the bank, and `for p in c` or `for (i, p) in enumerate(c)` can be used to iterate over the banks.

Combining Banks and Multiports

Banks of reactors may be combined with multiports (see MultiportToBank):

```
reactor Source {
    output[3] out:int;
    reaction(startup) -> out {=
        for(int i = 0; i < out_width; i++) {
            SET(out[i], i);
        }
    =}
}
reactor Destination(
    bank_index:int(0)
) {
    input in:int;
    reaction(in) {=
        printf("Destination %d received %d.\n", self->bank_index, in->value
    =}
}

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

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 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 ThreadedThreaded 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 {=
                SET(out, 42);
        =}
}
reactor Destination {
        input in:int;
        reaction(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.

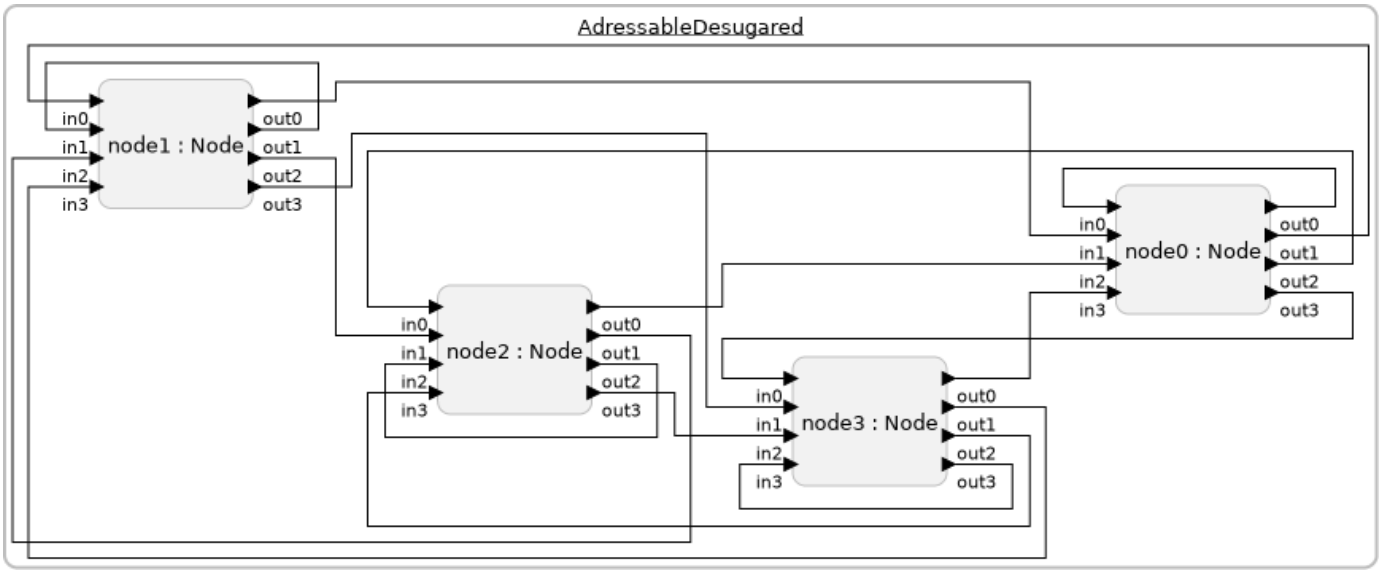
Interleaved Connections

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 Nth reactor instance within the destianation 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 FullyConnected_01_Addressable example, which is simplified below:

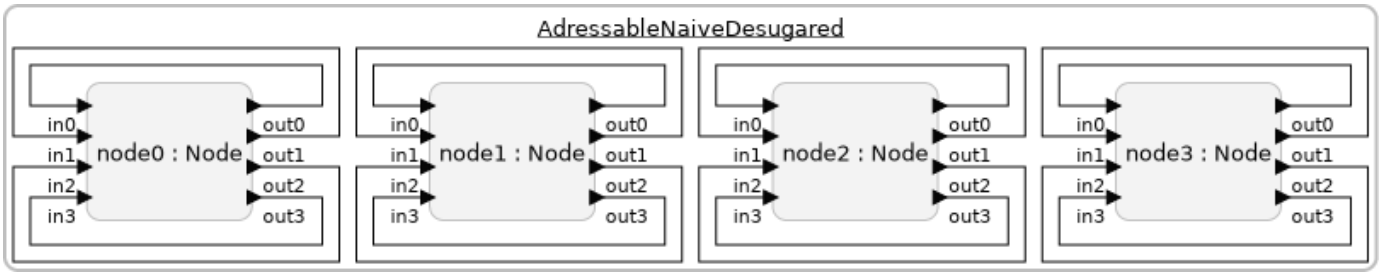
```
reactor Node(
    num_nodes: size_t(4),
    bank_index: int(0)
) {
    input[num_nodes] in: int;
    output[num_nodes] out: int;

    reaction (startup) -> out {=
        SET(out[1], 42);
    =}

    reaction (in) {=
        ...
    =}
}

main reactor(num_nodes: size_t(4)) {
    nodes = new[num_nodes] Node(num_nodes=num_nodes);
    nodes.out -> interleaved(nodes.in);
}
```

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. The 0-th channel will receive 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.

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 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 pattern and shows a desugared version constructed without banks or multiports:



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



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

Distributed Execution

```
! The information in this page is outdated, and will be updated soon.
```

NOTE: This describes a highly preliminary capability to map pieces of a Lingua Franca program to different machines. This capability is very much under development. This capability has been tested on MacOS and Linux, but not yet on Windows. Volunteer to do that and update these instructions?

A Lingua Franca program can be separated into separate programs called **federates** that communicate with one another. The federates will execute in separate processes and even on separate machines. They can be distributed across networks and can even be written in different target languages.

There is always one federate named **RTI**, for **runtime infrastructure** that coordinates startup and shutdown and may, if the implementation is centralized, mediate communication. The RTI needs to be compiled and installed separately on the system before any federation can execute. The instruction on how to do so can be found here.

Other than the RTI, if there are *n* federates, then the code generator will generate *n* separate programs with names of the form *Name_Federate*, where *Name* is the name of the top-level Lingua Franca file (without the .lf extension) and *Federate* is the name of the federate reactor. The code generator also produces a shell script that lauches all the federates and the RTI and a second shell script that distributes the generated code for the federates (not the RTI) to the specified machines and compiles the code on that machine.

Minimal Example

A minimal federated execution is specified by using the **federated** keyword instead of **main** for the main federate. An example is given in example/C/Federated/HelloWorld/HelloWorld.lf, which looks like this:

```
target C;
reactor MessageGenerator { ... }
reactor PrintMessage { ... }

federated reactor DistributedHelloWorld {
    source = new MessageGenerator();
    print = new PrintMessage();
    source.message -> print.message;
}
```

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 example/C/Federated/HelloWorld/HelloWorld.lf, the following three programs will appear in the `bin` directory:

- HelloWorld
- HelloWorld_source
- HelloWorld_print

The root name, *HelloWorld*, is the name of the .lf file from which these are generated. The suffixes "_source" and "_print" 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/HelloWorld`, which is a `bash` script that launches the other three programs. Alternatively, you can manually execute the RTI and the federate programs by starting them on the command line in any order.

In addition, one more `bash` shell scripts may be generated:

- HelloWorld_distribute.sh

This script is generated if any of the two federates, or the RTI are specified to be run on a remote machine (see below for how to do that). This script will copy the source files for the relevant program (but not the RTI) to the remote machine and compile them there. The RTI needs to be separately installed on the remote machine.

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

```
source.out -> print.in after 200 msec;
```

then the timestamp at the receiving end will be incremented by 200 msec compared to the timestamp at the sender (see example/C/Federated/HelloWorld/HelloWorldAfter.lf).

The preservation of timestamps across federates implies some constraints (see physical connections below for a way to avoid these constraints). How these constraints are managed depends on whether you choose **centralized** or **decentralized** coordination.

Containerized Execution

FIXME: Point to /docs/handbook/containerized-execution

FIXME: Here is an test that imports an existing federated test with the addition of a docker: true flag in the target property of the test. This test will automatically run in multiple Docker containers (one for the RTI and one for each federate) in our CI.

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 `print` federate has an event with timestamp *t* that it wants to react to (it is the earliest event in its event queue or it is a **physical action** that just triggered), 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 `print` 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 `source` is *t*. This message cannot be sent before the local clock at `source` reaches *t* and also cannot be sent before the RTI grants to `source` a time advance to *t*. Since `source` has no federates upstream of it, the RTI will always grant it such a time advance.

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 `print` federate will not see the message from `source` before physical time $t + L$, where this physical time is measured by the physical clock on `source`'s host. If that clock differs from the clock on `print`'s host by E, then `print` 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 `print` federate), therefore, will depend on the relationship between a and $E + L$. If, for example, $E + L > a$, then federate `print` will lag behind physical time by at least $E + L - a$.

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

If $a > E + L$, then the existence of this communication does not cause `print`'s logical time to lag behind physical time. This means that if a **physical action** appears at `print`, the RTI will be able to immediately grant a time advance to `print` to the timestamp of that physical action. However, if $a < E + L$, then the RTI will delay granting a time advance to `print` 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 set **after** on the connection to that federate to be 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 problem if you are using centralized coordination.

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
```

Centralized coordination works as described above, where the advancement of time at each federate is regulated by the RTI. In order for the RTI to be able to safely grant a time advance to a federate, it is also necessary for all messages to that federate to go through the RTI. The RTI, therefore, can easily become a bottleneck.

An alternative is **decentralized** coordination, which uses a technique called PTIDES:

```
coordination: decentralized
```

This technique has also been implemented in Google Spanner, a globally distributed database system. In decentralized coordination, each federate has a **safe-to-process** (**STP**) offset. In decentralized coordination, 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. This will work fine under the assumption that **every** logical connection between federates has a sufficiently large `after` clause. That is, the value of the logical delay must exceed the sum of the clock synchronization error *E*, the network latency bound *L*, 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**. See for example example/C/Federated/HelloWorld/HelloWorldDecentralized.lf.

Of course, this assumption can be violated in practice. Analogous to a deadline violation, Lingua Franca provides a mechanism for handling such a violation that is called a `tardy` handlers as done in example/C/Federated/HelloWorld/HelloWorldDecentralized.lf. (example/C/Federated/HelloWorld/HelloWorldDecentralized.lf). The pattern is:

```
reaction(in) {=
    // User code
=} tardy {=
    // Error handling code
=}
```

If the timestamp at which this reaction is to be invoked (the value returned by `get_current_tag`) cannot match the timestamp of an incoming message `in` (because the current tag has already advanced beyond the intended tag of `in`), then the `tardy` handler will be invoked instead of the normal reaction. Within the body of the tardy 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. If no tardy handler is provided at any reaction triggered by an input from

another federate, then the normal reaction will be invoked at the earliest feasible logical time greater than or equal to the intended logical time of the message.

One option available to the code is to increase the STP. This can be done simply by equipping a federate with a parameter of type **time** named `STP`. See for example example/C/Federated/HelloWorld/HelloWorldDecentralizedSTP.lf. This can be done as follows:

```
import PrintMessageWithDetector from "HelloWorldDecentralized.lf"
reactor PrintMessageWithSTP(STP:time(10 msec)) extends PrintMessageWithDete
```

Notice that the only override in `PrintMessageWithSTP` is the addition of an `STP` parameter.

The LF API provides two functions that can be used to dynamically adjust the STP:

```
interval_t get_stp_offset();
void set_stp_offset(interval_t offset);
```

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

FIXME: The discussion of cycles in the remainder of this section needs to be revisited with pointers to newer examples.

Now suppose that if there are cycles in the communication between federates. For example, in addition to the above connection, suppose we also have a connection going in the opposite direction:

```
print.out -> count.in after 100 msec;
```

Now we potentially have a very big problem. The physical clock at `print` has to lag behind physical time by at least $E + L - 200$ msec, and the physical clock at `count` has to lag behind physical time by at least $E + L - 100$ msec. The latter of these means that `count` cannot send a message with timestamp t until its local clock exceeds $t + E + L - 100$ msec. If $E + L - 100$ msec > 0, then this additional lag increases the required lag at `print`, which will need to lag behind physical time now by $E + L - 100$ msec $+ E + L - 200$ msec. If this number is positive, then the lag required at `count` will have to be again increased, which will then cause this number to again increase, and so on until the required lag is infinite at both federates. Thus, a cycle between two federates is **infeasible** if $2E + 2L - a_1 - a_2 > 0$, where a_1 and a_2 are the **after** values of the two connections. More generally, the sum of $E + L - a_i$ over all connections i in a cycle must be less than or equal to zero. Otherwise, decentralized coordination will fail and finite STP will lead to tardy messages. Centralized coordination can be used instead if the program really must be this way.

The bottom line is that if there are cycles in your federation and/or you have physical actions in federates that receive network messages, it is wise to specify **after** to be larger than the sum of the greatest expect clock synchronization error E and the greatest expected network latency L.

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 (see example/C/Federated/HelloWorld/HelloWorldPhysical.lf:

```
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. **FIXME**: Windows?

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. Instructions about installation of RTI can be found here.

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.

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.

FIXME: Not implemented yet: If the IP address or hostname does not match the local machine on which code generation is being done, ...

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.

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 incomming connections from other federates.

If any federate (or the RTI) 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.

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: In the above example, `localhost` is used. This is the default if no address is specified. Using `localhost` specifies that the generated programs should establish connections only with processes running 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.

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`.

Future Work

The RTI can also play the role of **auth**, an authentication and authorization server that ensures that only the generated programs can establish connections with each other and that their communication is encrypted, as explained in the Security section below.

Currently, the threads option is same on all federates. We need a mechanism to customize this parameter by federate.

Security

In addition to generating a program for each host, the code generator could generate configuration files for a program called **auth** designed to run on the first host that is preconfigured to provide authentication and authorization to each of the other generated programs together with encryption keys that are used for communicating between them. The auth program should be started first since non of the other generate programs will be able to authenticate without it.

The auth program, written by Hokeun Kim, comes from https://github.com/iotauth/iotauth and provides "locally centralized, globally distributed" authentication and authorization. Papers describing this work can be found here: [IoTDI '17], [FiCloud '16] [IT Professional '17'].

Protobufs

Communication between hosts can only be accomplished on channels where the message types are either language primitives or Protobufs. All other datatypes will be reject at code generation time.