Issues in Concurrent Language Design

John Reppy
AT&T Labs Research

November 1996

Assumptions

- My focus is on *concurrency*, not *parallelism* or *distributed programming*.
- Motivation is concurrency in user interfaces and concurrency in distributed systems.
- Higher-order sequential language: functions as values, data abstraction, polymorphism.

What is important?

- Robustness and correctness.
- Expressiveness.
- Modularity.
- Performance.

Synchronization and communication

The choice of synchronization and communication mechanisms is the most important design choice in a concurrent language.

- Should these be independent or coupled?
- What guarantees should be provided?

Synchronization *(continued ...)*

There are a range of mechanisms found in concurrent languages:

- Shared memory (locks and condition variables)
- Synchronous memory (I-structures and M-structures)
- Asynchronous message passing (buffered channels)
- Synchronous message passing (blocking send)
- RPC (aka extended rendezvous)

Design point:

Shared-memory is a poor programming model

- Although one can write very efficient programs this way, the model does not promote correctness.
- It requires defensive programming (protect your data/code from interference), without compiler support.
- Shared-memory primitives do not fit well with a value-oriented programming style.

Message ordering

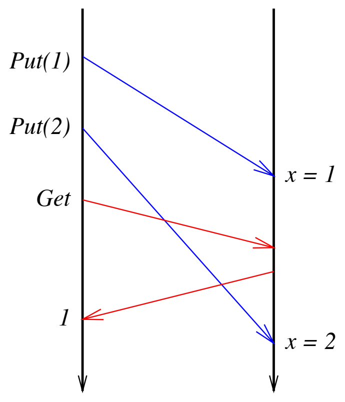
Some recent designs have proposed treating buffered channels as multisets (instead of as queues).

This has implementation advantages in distributed settings, and allows an easy *undo* mechanism for communications.

Design point:

FIFO ordering on messages is good

There are very few interprocess interactions that do not require at least a FIFO ordering on messages.



Transparent distribution

Some people argue that we should implement concurrent languages on distributed systems in a *transparent* fashion — i.e., local and remote communication should look the same.

Design point:

Transparent distribution is problematic

- Remote operations have high latency; local ones do not.
- Transferring large data structures locally is done by pointer copying; remote transfer requires much more work.
- Remote systems/links may fail, but concurrent languages do not provide a model these kinds of failures.

Extended rendezvous

Some languages provide request/reply as the communication mechanism (Ada, Concurrent C).

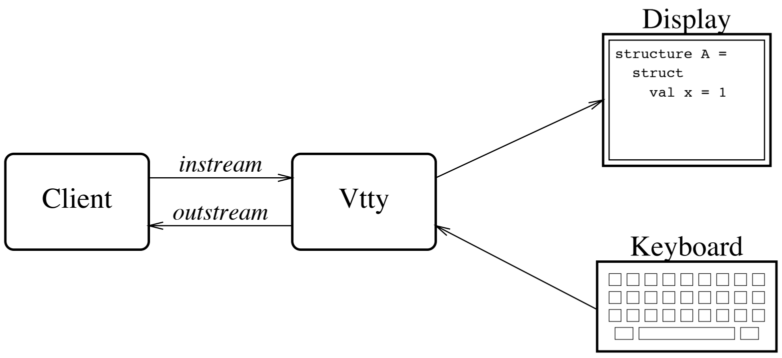
Design point:

Extended rendezvous is too much

- Extended rendezvous is asymmetric and does not support data-flow networks.
- It is easy to implement extended rendezvous on top of message-passing.

Selective communication

In a language with blocking communication operations (e.g., `recv` and blocking `send`), it is useful to choose between a set of blocking operations.



Can solve this by union types and extra threads; in other examples, we may use special protocols.

Design point:

Selective communication is important

- It reduces the number of threads required.
- It promotes modularity, since specialized protocols may not compose.

Design point:

Synchronous communication is powerful

More specifically, the combination of blocking send and a choice operation provides a mechanism for attaining 2-way *common knowledge*.

Design point:

Abstraction is crucial to writing/maintaining correct software.

Unfortunately, most languages do not support synchronization/communication abstractions.

Design point:

Negative acknowledgements promote modularity

Negative acknowledgements are a mechanism that are unique to CML. They provide a way to implement abortable protocols as abstractions.

Concurrent ML

- Provides a uniform framework for synchronization: *events*.
- Event combinators for constructing abstract protocols.
- Collection of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels

  Plus I/O, timeouts, thread join, ...

`http://www.research.att.com/~jhr/sml/cml/`

Basic CML features:

```
type thread_id
type 'a chan
type 'a event

val spawn : (unit -> unit) -> thread_id

val channel : unit -> 'a chan
val recv    : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit

val recvEvt : 'a chan -> 'a event
val sendEvt : ('a chan * 'a) -> unit event

val choose   : 'a event list -> 'a event
val guard    : (unit -> 'a event) -> 'a event
val wrap     : ('a event * ('a -> 'b)) -> 'b event
val withNack : (unit event -> 'a event) -> 'a event

val sync     : 'a event -> 'a
val select   : 'a event list -> 'a
```

CML Linda

The *Linda* family of languages use *tuple spaces* to organize distributed computation.

A tuple space is a shared associative memory, with three operations:

output adds a tuple.

input removes a tuple from the tuple space. The tuple is selected by matching against a *template*.

read reads a tuple from the tuple space, without removing it.

The CML interface is:

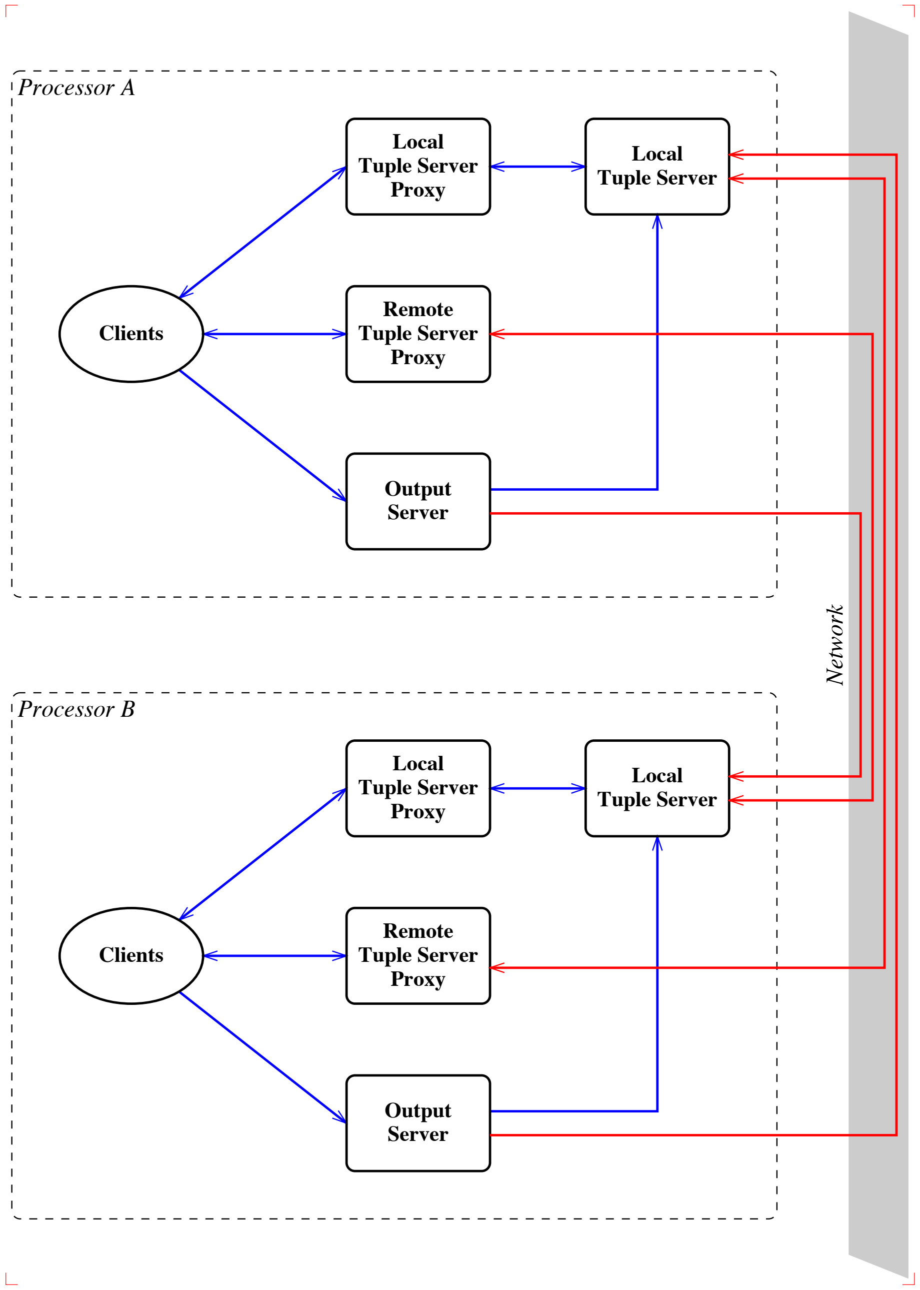
```
val output : (ts * tuple) -> unit
val inEvt  : (ts * template) -> value list event
val rdEvt  : (ts * template) -> value list event
```

CML Linda *(continued ...)*

There are two ways to implement a distributed tuple space:

- *Read-all, write-one*
- *Read-one, write-all*

We choose read-all, write-one. In this organization, a `write` operation goes to a single processor, while an `input` or `read` operation must query all processors.


Processor A
Local Tuple Server Proxy
Local Tuple Server
Clients
Remote Tuple Server Proxy
Output Server
Network
Processor B
Local Tuple Server Proxy
Local Tuple Server
Clients
Remote Tuple Server Proxy
Output Server

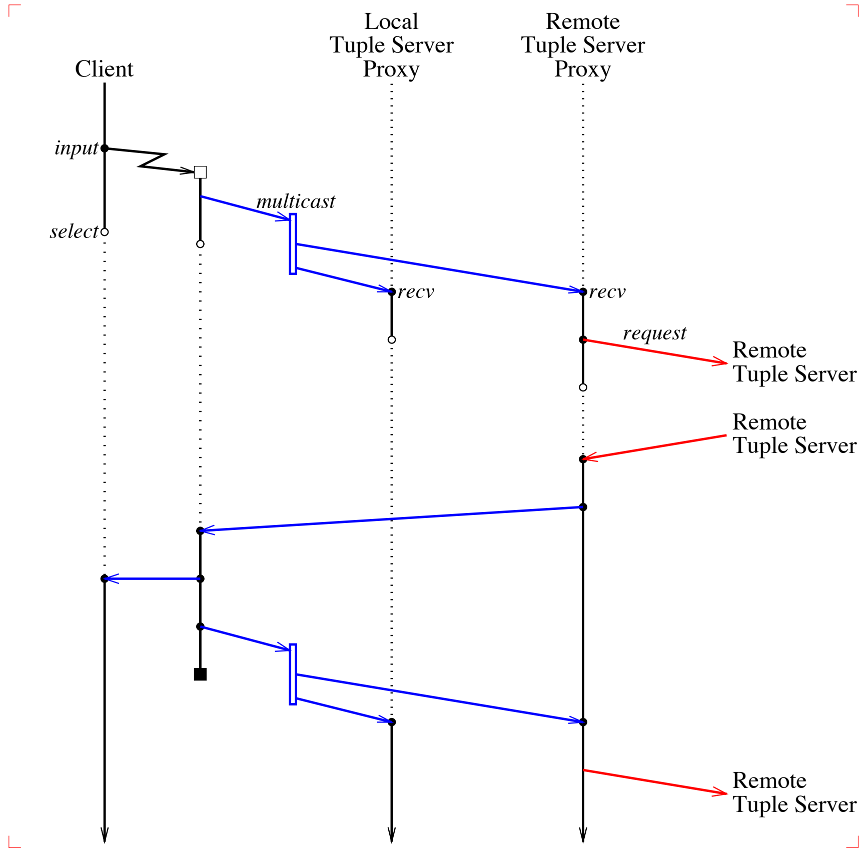
CML Linda *(continued ...)*

The `input` protocol is complicated:

1. The reader broadcasts the query to all tuple-space servers.
2. Each server checks for a match; if it finds one, it places a *hold* on the tuple and sends it to the reader. Otherwise it remembers the request to check against subsequent `write` operations.
3. The reader waits for a matching tuple. When it receives a match, it sends an acknowledgement to the source, and cancellation messages to the others.
4. When a tuple server receives an acknowledgement, it removes the tuple; when it receives a cancellation it removes any hold or queued request.

CML Linda *(continued ...)*

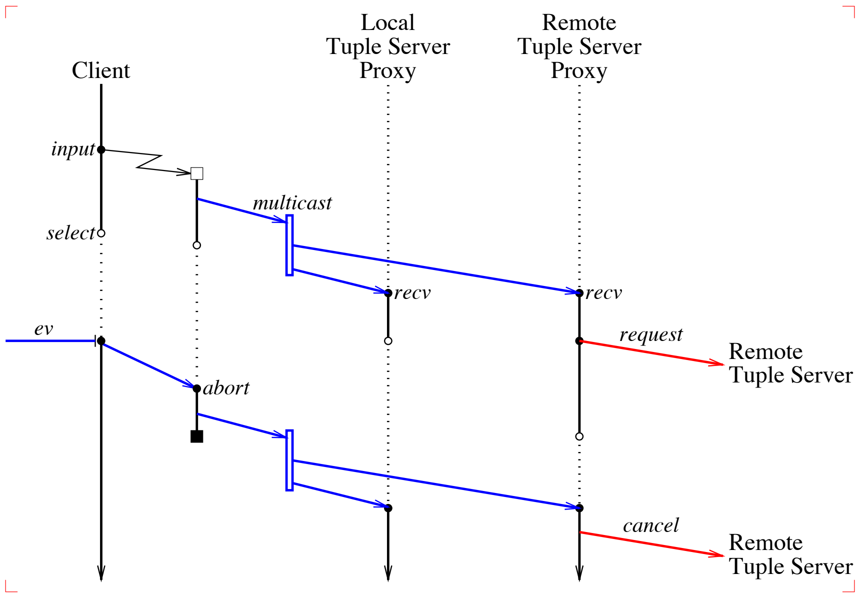
Here is an example of a successful input operation:



Note that we must confirm that a client accepts a tuple, before sending out the acknowledgement.

CML Linda *(continued ...)*

We use negative acknowledgements to cancel requests, when the client chooses some other event.



Conclusions:

Concurrent programming is hard, and concurrent languages should be designed to provide as much support as possible.

This means:

- Avoid asynchronous access to shared state.
- Provide strong synchronization guarantees.
- Provide support for application specific abstractions.