Fact-based Inter-Process Communication Primitives for Programming Distributed Systems

Robbert van Renesse, Department of Computer Science Cornell University

Category: Representation

The following position paper describes a new Inter-Process Communication (IPC) primitive that is designed to make it easier to program distributed algorithms. It is largely based on my experience in implementing algorithms such as distributed consensus, leader election protocols, replication protocols, and so on. Subject to your evaluation of my proposal, I would be happy to present this idea at the workshop.

IPC allows processes to share information and to synchronize actions. Essentially, there are two classes of IPC: message channels (MC) and shared memory (SM). MC has processes communicate send and receive messages, while SM allows processes to share data directly while synchronizing using such primitives as mutexes and condition variables. In distributed systems, where processes are physically separated, MC is dominant as efforts to support the SM paradigm have not been successful. Examples of SM include TCP connections, RPC, and pub/sub. The MC and SM paradigms are duals in that one can be implememted using the other, but they also each have their advantages and disadvantages when compared with one another.

It is useful to consider how distributed algorithms such as replication, leader election, parallel algorithms, apply IPC. Typically it has much to do with progress: in order for some process to be able to make a transition, it needs to know that one or more other processes have reached a particular milestone, and some data associated with that milestone. For example, a new leader in Paxos needs to know that a quorum of acceptors have progressed to its proposed ballot and it needs to know what the highest accepted proposals from those acceptors are. Many if not all distributed algorithms can be cleanly expressed this way: as a collection of transition specifications that specify under which conditions they are enabled and what state they need from other processes. Note the similarity to knowledge-based programming.

While the SM paradigm seems the best fit for this model of distributed algorithms, the paradigm is hard to make efficient, secure, and scalable in a physically distributed system. Also, it is notoriously error-prone as programmers are having difficulty utilizing the synchronization primitives correctly. The MC paradigm can be used instead but is awkward and error-prone as well—it requires the programmer to figure out which processes should send which data to which destinations at which times in order to ensure that recipients of this data can make progress. Sometimes messages are lost if the receiver starts execution after the sender has started sending messages to it. (The familiar "send-and-pray" semantics of connectionless or non-blocking messaging primitives is one example.) Often needless information is sent as more recent information makes old messages obsolete. Using Paxos again as an example: in the stream of values that acceptors accept, only the most recent one is of interest. But most MC implementations will carefully deliver each and every one, delaying delivery of the important information until all obsoleted information has been delivered as well. This leads to wasting resources, potential deadlock situations due to flow control leading to deadly embrace, and also obfuscates how the algorithms work.

We propose Fact-Based IPC, a new class of IPC, that that tries to combine the best features of SM and MC. From SM it inherits direct access to and synchro-

nization on state rather than providing a stream of state updates, while from MC it inherits an efficient implementation over the existing physical infrastructure. The concept is that processes publish *facts*, which are information about milestones they have reached, and subscribe to new facts. The IPC interface is similar to topic-based pub/sub-based messaging, but there are several important semantic differences. The (familiar) interface is as follows:

- `publish`(*topic*, *fact*)
- `subscribe`(*topic*, *upcall*)

The interface requires that the *fact* type for a particular topic is totally ordered, and those facts will be delivered in order. (Any data can be made totally ordered by tagging it with a sequence number, but often times facts such as ballots are totally ordered already.) Given a stream of facts on some topic, only the highest, most recent fact need be delivered eventually, while older facts can be dropped. Also different from pub/sub messaging, if no more facts are published but some process later subscribes, it will eventually receive the most recent fact (assuming both publisher and subscriber are correct). These semantics are similar to the anti-entropy style of gossip protocols, but the underlying implementation can be anything.

There is also a control interface that controls routing of facts for a particular topic. For example, Paxos acceptors subscribe to ballots and to new proposals from leaders. When the leader publishes one of these, it is transmitted to all subscribers, and the underlying communication layer will continue retransmission until either acknowledged or another fact renders it obsolete.

For some topics, it will be the publishers that actively try to push new facts to the subscribers. For example, Paxos leaders publish new ballots and push these to acceptors as acceptors do not necessarily know what the set of leaders is. Old ballots are automatically dropped from the transmission queue. For other topics, it will be the subscribers that actively poll the publishers. For example, leaders and learners both subscribe to acceptors accepting pvalues and poll for these facts. New subscribers, as well as subscribers that suffered communication loss due to a network partition or having been temporarily shutdown (*e.g.*, due to a user closing a laptop), will continue to poll publishers to receive facts they have missed. All this is invisible to the core application programmer, but can be managed through the control API.

The hope is that Fact-based IPC will simplify distributed programming and make it easier to reason about safety and liveness. The argument for this is that the paradigm allows the programmer to clearly specify transitions and under which conditions they are enabled without having to worry much about how these conditions are discovered.