The Network as a Language Construct

Tony Garnock-Jones[1], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

[1]Northeastern University, Boston, Massachusetts, USA
[2]Indiana University, Bloomington, Indiana, USA

Abstract. The actor model inspires several important programming languages. In this model, communicating concurrent actors collaborate to produce a result. A pure actor language tends to turn systems into an organization-free collection of processes, however, even though most applications call for layered and tiered architectures. To address this lack of an organizational principle, programmers invent design patterns.

This paper investigates integrating some of these basic patterns via a programming language construct. Specifically, it extends a calculus of communicating actors with a "network" construct so that actors can conduct scoped, tiered conversations. The paper then sketches how to articulate design ideas in the calculus, how to implement it, and how such an implementation shapes application programming.

1 Organizing Squabbling Actors

Hewitt's actor model [16] presents computation as a collaboration of concurrent and possibly parallel agents. Collaboration necessitates communication, and all communication among actors happens by message passing. The resulting separation of actors isolates resources and thus prevents conflicting use due to competing activities. Several programming languages and frameworks use the actor model as a design guideline, most prominently Erlang [8] and Scala [14].

Like the λ-calculus, the actor model is an elegant foundation for language design but fails to scale to real systems. Hence, a pure actor language turns programs and systems into organization-free "soups of processes." More precisely, the model provides no organizational principle that helps programmers arrange collections of actors into a layered or tiered architecture; also out of scope is the management and monitoring of actors via actors. Similarly, the model does not support common idioms of communication, such as multi-cast messaging, sessions, or connections. Finally, it ignores exceptions and errors, meaning it does not deal with partial failures.

Implementations of the actor model meet programmer demand for organizational principles with libraries whose APIs and protocols realize appropriate design patterns. Many such APIs hide a mini language that deserves the same kind of focused study that proper linguistic features earn. In this paper, we explain the *network* as such a hidden language feature. Our central innovation is the *Network Calculus*, which explains how to equip a given programming language with networks. Our prototype implementation of the calculus, *Marketplace*, illustrates the potential of the network as a language construct.

2 Our Model of Actors

While Agha et al. [1] present an elegant operational semantics as a verification framework for imperative actors, our goal is to create a calculus of actors to articulate a language design idea. Specifically, we wish to show how to construct an actor language from an *arbitrary* base language via the addition of a fixed communication layer. To this end, we make the state of actors explicit, require their specification as a state-transition function, and demand that they interact exclusively via messages—not effects. This strict enforcement of the message-passing discipline does *not* prevent us from using an imperative base language, as long as its effects do not leak. In other words, the base could be a purely functional language such as Haskell, a higher-order imperative language such as Racket, or an object-oriented language such as JavaScript.[1]

The abstract syntax of our calculus is straightforward:

$$
\begin{array}{lll}
C = [\overline{\alpha}\,\overline{A}] & C_Q = [\cdot\,\overline{A_Q}] & \text{(Actor Configurations)} \\
A = x : \Sigma & A_Q = x : \Sigma_Q & \text{(Actors)} \\
\Sigma = \overline{a} \lhd B & \Sigma_Q = \cdot \lhd B & \text{(Actor States)} \\
B = f ; u & & \text{(Simple Behaviors)} \\
a = \alpha \mid A & & \text{(Actions)} \\
\alpha = \langle x, v \rangle & & \text{(Events)} \\
v = u \mid x \mid v, v & & \text{(Message Values)}
\end{array}
$$

We use $\overline{p}$ to denote a queue of ps. The xs in this grammar are drawn from an unspecified set of names or atoms; u ranges over base language values.

An actor configuration C consists of some actors $\overline{A}$ and a queue $\overline{\alpha}$ of pending events. An actor is a named (x) state that combines a behavior B with a queue $\overline{a}$ of pending actions. A behavior B pairs a function f with a state value u, both from the base language. We use A_Q to denote the set of *quiescent* actors, i.e., those with an empty queue of pending actions; a quiescent configuration C_Q has no pending events and all its actors are quiescent. Our actors may perform one of two actions: send a message or create another actor. The latter is specified as $x : \Sigma$, i.e., a complete actor, while $\langle x, v \rangle$ denotes a request to send message v to the actor named x. On receipt of a message, actor x computes the actions $\overline{a}$ it wishes to perform.

This response computation makes up the complete interface between the base language and the communication layer.[2] The interface consists of an interp_0 function, which interprets an actor-level event α and yields actor-level actions $\overline{a}$:

$$\mathit{interp}_0 : f \times \alpha \times u \to \overline{a} \times u$$

[1] In fact, each base actor could in principle use a different language, turning the network calculus into a semantics of middleware.

[2] The traditional actor model includes a `become` primitive, updating an actor's code and its state simultaneously. Such a primitive would require that the interpretation function delivers actions, a state, and a state-transformation function, i.e., $f \times \overline{a} \times u$.

The base language itself must also include facilities for analyzing and constructing representations of network-level events and actions, respectively.

We can now formulate the dispatch rule for communicating actors:

$$\frac{\overline{A_Q \xrightarrow{\alpha} A'}}{[\alpha\overline{\alpha}_0\ \overline{A_Q}] \longrightarrow [\overline{\alpha}_0\ \overline{A}']}\ \text{(dispatch)}$$

To keep the interactions between the base and the network simple, the dispatch rule fires only when all the configuration's actors are quiescent. It relies on an event-indexed family of relations $\xrightarrow{\alpha}$ that dispatch events α to actors:

$$\frac{\mathit{interp}_0\ f\ \alpha\ u = (\overline{a}, u')}{x : \cdot \triangleleft f; u \xrightarrow{\alpha} x : \overline{a} \triangleleft f; u'}\ \alpha = \langle x, v\rangle \qquad \frac{}{x : \Sigma \xrightarrow{\alpha} x : \Sigma}\ \alpha \neq \langle x, v\rangle$$

Since the dispatch rule adds actions to an actor's queue, we need two additional rules to interpret these actions:

$$[\overline{\alpha}\quad \overline{A_Q}\ (x : \langle y, v\rangle\overline{a} \triangleleft B)\ \overline{A}] \longrightarrow [\overline{\alpha}\langle y, v\rangle\quad \overline{A_Q}\ (x : \overline{a} \triangleleft B)\ \overline{A}] \qquad \text{(send)}$$
$$[\overline{\alpha}\quad \overline{A_Q}\ (x : A_{new}\overline{a} \triangleleft B)\ \overline{A}] \longrightarrow [\overline{\alpha}\quad \overline{A_Q}\ (x : \overline{a} \triangleleft B)\ \overline{A}\ A_{new}] \qquad \text{(spawn)}$$

Due to their syntactic constraints, dispatch and interpretation alternate, and messages sent by any given actor are received by peers in order.

While *spawn* is semantically straightforward, users of the model have a pragmatic hurdle to overcome if they wish to ensure *uniqueness* of actor names in a configuration. They may choose to use a "name-factory" service, to preallocate names, or any other of a wide range of appropriate strategies. Unique naming in a distributed setting is a well-known thorny issue, and it is one of our motivations in separating actor naming and addressing from actor identity below.

In our calculus, actors do not block; they remain responsive to inputs. Traditional behaviors such as "nested receive" and mailbox filtering are still expressible using well-known techniques [18].

Our actor calculus satisfies basic correctness theorems. First, the communication layer does not add any errors. Second, it is deterministic.

Theorem 1 (Soundness). *If interp_0 is total, an actor configuration C is either quiescent or there exists C' such that $C \longrightarrow C'$.*

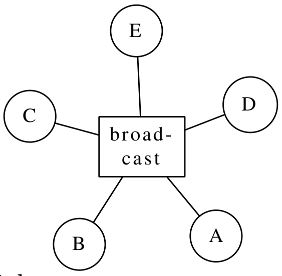
Proof (Sketch). The lemma is a reasonably standard "progress lemma" and follows from a conventional proof approach [32]. □

Theorem 2 (Determinism). *For any actor configuration C, there is at most one C' such that $C \longrightarrow C'$ (modulo systematic actor renaming).*

Proof (Sketch). The lemma is a conventional diamond lemma and follows from an inspection of all possible critical pairs in the reduction relation. □

3 Making Networks a Proper Part of the Language

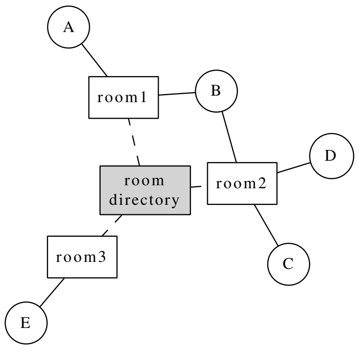
The actor model of computation comes with one special, built-in network that connects the actors to each other. Both our experience and the literature [6,11,35] lead us to argue, however, that programmers must be able to create and manage recursively-nestable networks. To make this point, we sketch and analyze the implementation of a chat room server, a typical example. We assume that users access this chat room over TCP via a `telnet`-like client and that the chat room broadcasts each "line" from one user to every other user. As users connect, these new connections are announced to the signed-up users, and the list of already-connected peers is sent to the new user. Disconnections are announced in a similar fashion.



A natural starting point is to create one actor per connecting user. Unfortunately the point-to-point messaging in an actor network conflicts with the desire to *broadcast* messages among users. One option is for each actor to maintain a list of peers, but synchronizing so much state becomes challenging as the number of participants grows. A more scalable, idiomatic option is to reify the medium of communication as a "broadcasting" actor, shown at right.

Our service now involves two different classes of actor: one for relaying to and from connected users and one for mediating the interactions of the former. The system cleanly divides responsibilities among the two. Each "relay" actor registers with the "chatroom" actor, relays messages from the chatroom to the associated TCP socket, and parses incoming utterances from the TCP socket, converting them into chatroom messages. The chatroom actor, for its part, must manage a directory of active relay actors, announce comings and goings, and broadcast chat messages received from the relays.

Both kinds of parties must detect failure in other actors. A failing relay actor should be treated as if the user had requested disconnection, causing both the closing of the associated TCP socket and an announcement of the departure of the user to the remaining users. Similarly, if the chatroom actor fails, each relay actor should take some emergency action such as announcing the problem and presumably closing its connection.



Now imagine that the service supports several named chat rooms and that connecting users may join any number of rooms. This refined design calls for a directory mapping room names to actor addresses. Again, it is natural to implement this service as an actor. This "room directory" actor maintains a directory of available chat rooms and responds to room lookup requests from relay actors. Each chatroom actor now registers itself with the room directory as it is created and relay actors query the room directory as users ask to join individual chatrooms. Finally, all three types of actor are now committed to monitoring the health of other actors to maintain their own state.

As simple as it is, the scenario exemplifies a number of classic patterns in actor architectures that internalize concepts from distributed object systems:

– dynamic *naming service*, illustrated by directory and chatroom actors;
– a *multicast medium*, implemented in the chatroom actor and also known as a data distribution service [23] or a publish/subscribe broker [9];
– dealing with *partial failure* [31]; each actor must detect and handle failures in the assembled system as part of its regular duties.

Furthermore, we can identify common patterns from layered architectures [27]:

– Members of each layer communicate using a layer-specific protocol. As messages cross layers, actors translate them from one protocol to another. In our example, relay actors translate from TCP to chat messages.
– Layers isolate components. Explicit relaying and message transformation protects services from external requests. For the chat service, the relay actors make up the periphery, protecting chat rooms and room directories.

The chat room scenario is simple but not special. Programmers re-implement naming services and multicast over and over. They ensure isolation and protection for layers of communication in actor systems. And they equip actors with code to detect and signal partial failures. This happens regardless of whether the system is sequential, concurrent or parallel, distributed or monolithic.

In response, we supply these services via a novel linguistic construct, dubbed "network" because it internalizes networking-style programming into actor systems. A network is a communications medium and a resource container [6,35] offering *naming*, *delivery*, *relaying*, and *multiplexing* services to its clients and enabling them to monitor the coming and going of their peers.

4 The Network Calculus

To address the design concerns raised above, we experimented with three generations of actor middleware and a number of full-scale applications. Here we distill our experience into a calculus of networks, which we consider a tool for language re-design. This section presents the calculus, again as a layer atop a sequential language, and sketches some variations. The next two illustrate how to "code" in the calculus and how to use it as a guide to language design.

The Calculus We extend our actor calculus to the Network Calculus (NC) in three syntax design steps: (1) we turn the communications substrate into a linguistic construct; (2) we generalize from point-to-point messaging to broadcasts; and (3) we signal routing changes as events alongside regular messages.

The first step is to promote configurations to behavior status:

$$\begin{array}{llll} B = f;u \mid C & B_Q = f;u \mid C_Q & B_I = f;u \mid C_I & \text{(Behaviors)} \\ C = [\overline{\alpha}\,\overline{\pi}\,\overline{A}] & C_Q = [\cdot\,\overline{\pi}\,\overline{A_Q}] & C_I = [\cdot\,\overline{\pi}\,\overline{A_I}] & \text{(Configurations)} \end{array}$$

This change allows programs to spawn entire networks recursively as actors. Multiple networks can thus exist side-by-side and nested within other configurations. Each actor, except for the root configuration, is contained in a configuration.

The second step introduces a form of publish-subscribe messaging that allows both point-to-point messaging and broadcasting. Actors thus no longer come with names but *subscriptions* $\overline{\pi}$ describing their interests, and a network maintains a set of subscriptions $\overline{\pi}$, describing the interests of its environment:

$$\begin{array}{llll} A = \overline{\pi} : \Sigma & A_Q = \overline{\pi} : \Sigma_Q & A_I = \overline{\pi} : \Sigma_I & \text{(Actors)} \\ \Sigma = \overline{a} \triangleleft B & \Sigma_Q = \cdot \triangleleft B & \Sigma_I = \cdot \triangleleft B_I & \text{(Actor States)} \end{array}$$

In addition to quiescent configurations C_Q and actors A_Q, we now distinguish *inert* variations: $C_I \subset C_Q \subset C$ and $A_I \subset A_Q \subset A$. While a quiescent configuration has merely emptied its *local* queues, inert variants have empty queues at all levels. In other words, inert actors or networks are waiting for *external* events.

A subscription is an expression of interest in certain messages. Messages come in two, symmetric varieties: $\langle v \rangle$, an ordinary data-bearing message, and (v), a *feedback* message that travels along routes in the opposite direction to ordinary messages. Feedback messages are useful for flow-control and acknowledgement signalling. A message may also be prefixed by $i \geq 0$ downward arrows $\downharpoonleft$, indicating that it should be relayed out to the ith-innermost containing network:

$$\begin{array}{ll} m = \langle v \rangle \mid (v) \mid \downharpoonleft m & \text{(Messages)} \\ v = u \mid x \mid v, v & \text{(Message values)} \end{array}$$

Messages carry binary trees[3] of values from the base language and atoms x from an unspecified set.

Equipped with messages, we can describe subscriptions:

$$\begin{array}{ll} \pi = \langle p \rangle_n \mid (p)_n \mid \downharpoonleft \pi & \text{(Subscriptions)} \\ p = u \mid x \mid p, p \mid \star & \text{(Message patterns)} \end{array}$$

Temporarily ignoring the subscripts, the syntax of a subscription specifies a tree over values and atoms and the wildcard $\star$. Semantically, a subscription π is an expression of interest in messages:

– $(p)_n$, a subscriber pattern specifies interest in receiving ordinary data messages and sending feedback messages,
– $\langle p \rangle_n$, a publisher pattern expresses an interest in receiving feedback messages and sending ordinary messages.

[3] Our choice of trees allows a convenient representation of arbitrary structured data along with a simple definition of pattern-matching. Many other suitable choices exist.

Just as for message transmission, prefixing a subscription with i downward arrows ($\downharpoonleft \pi$) indicates the subscription pertains to the ith-innermost containing network. A network aggregates subscriptions to form its routing table.

The third design step adds a novel form of event, a *routing event* $\overline{\pi}$.[4] Actions are extended in an analogous manner to match the new class of events:

$$\begin{aligned} \alpha &= m \mid \overline{\pi} && \text{(Events)} \\ a &= \alpha \mid S && \text{(Actions)} \\ S &= \overline{\pi} : \overline{a} \lhd f; u \mid \cdot : \cdot \lhd [\cdot\ \cdot\ (\cdot : \overline{S} \lhd \cdot; \cdot)] && \text{(Spawnable actors)} \end{aligned}$$

Every time a network's routing table changes, it sends routing events $\overline{\pi}$ to its actors describing its current aggregate routing table. Each actor's view of the routing table is then filtered by its own subscriptions, and in particular by the *observer level* (subscript n) attached to each subscription: a subscription $(p_1)_n$ is only signalled to actors with matching subscriptions $\langle p_2 \rangle_m$ where both $m > n$ and $p_1 \mid\mid p_2$; see figure 1 for the definition of the matching functions $(\cdot \mid\mid \cdot)$. Actors interested in participating in conversations without observing other participants subscribe at level $n = 0$; those interested in observing participants use $n = 1$; those interested in observing observers use $n = 2$; and so on.

To formulate the semantics of the NC, we need to introduce the notion of a *spawnable actor* because spawning an entire network requires restrictions to preserve routing table consistency. A spawned network holds a list of spawn actions, which start the configuration's processes. They run within a special, primordial process. A newly spawned actor whose subscription set is non-empty is effectively assigned responsibilities from the very first moments of its existence.

Actor configurations evolve via two reduction relations: $\Sigma \longrightarrow \Sigma'$ encodes internal reduction steps toward inertness, and $A \xrightarrow{\alpha} A'$ informs actors of the event α. Our formulation of the semantics relies on the definitions in figure 1.

When a configuration's actors are quiescent, it sends the next queued event:

$$\frac{\overline{A_Q \xrightarrow{\alpha} A'}}{\overline{a} \lhd [\alpha \overline{\alpha}_0\ \overline{\pi}_1\ \overline{A_Q}] \longrightarrow \overline{a} \lhd [\overline{\alpha}_0\ \overline{\pi}_1\ \overline{A}']} \text{(dispatch)}$$

Dispatched events are matched against the active subscriptions of each actor. Events not matching are ignored,[5] while matching events are delivered:

$$\frac{}{\overline{\pi} : \Sigma \xrightarrow{\alpha} \overline{\pi} : \Sigma} \quad \alpha \mid\mid \overline{\pi} \text{ is undefined}$$

$$\frac{\mathit{interp}_0\ f\ (\alpha \mid\mid \overline{\pi})\ u = (\overline{a}, u')}{\overline{\pi} : \cdot \lhd f; u \xrightarrow{\alpha} \overline{\pi} : \overline{a} \lhd f; u'} \quad \alpha \mid\mid \overline{\pi} \text{ is defined}$$

$$\frac{\mathit{inject}(\alpha \mid\mid \overline{\pi})\ C = C'}{\overline{\pi} : \cdot \lhd C \xrightarrow{\alpha} \overline{\pi} : \cdot \lhd C'} \quad \alpha \mid\mid \overline{\pi} \text{ is defined}$$

---

[4] A routing event is, on the one hand, like "service presence" in XMPP [25] and, on the other hand, resembles route advertisements in distance-vector routing protocols [15].

[5] Inspired by the "discard" relation of Ene and Muntean's broadcast π-calculus [7].

$table : A \to \overline{\pi}$	Extracts the current subscription set from an actor
$lift : \overline{\pi} \to \overline{\pi}$	"Lifts" a subscription set by prepending $\downharpoonleft$ to each subscription
$drop : \overline{\pi} \to \overline{\pi}$	Almost-inverse of *lift*: removes $\downharpoonleft$ from subscriptions in a subscription set, omitting those lacking $\downharpoonleft$.
$\alpha \mid\mid \overline{\pi} : \alpha \times \overline{\pi} \rightharpoonup \alpha$	Filter/restrict an event by a subscription set
	$(v) \mid\mid \overline{\pi} = (v)$, if $\exists \langle p \rangle_n \in \overline{\pi}$ such that $v \mid\mid_v p$
	$\langle v \rangle \mid\mid \overline{\pi} = \langle v \rangle$, if $\exists (p)_n \in \overline{\pi}$ such that $v \mid\mid_v p$
	$\downharpoonleft m \mid\mid \overline{\pi} = \downharpoonleft m$, if $m \mid\mid drop(\overline{\pi})$
	$\overline{\pi}_1 \mid\mid \overline{\pi}_2 = (\pi_{11} \mid\mid \pi_{21}) \cdot\cdot (\pi_{11} \mid\mid \pi_{2m})(\pi_{12} \mid\mid \pi_{21}) \cdot\cdot (\pi_{1n} \mid\mid \pi_{2m})$ where $\pi_{11} \cdots \pi_{1n} = \overline{\pi}_1$ and $\pi_{21} \cdots \pi_{2m} = \overline{\pi}_2$
$\pi \mid\mid \pi : \pi \times \pi \rightharpoonup \pi$	Intersection of two subscriptions, respecting observer level
	$(p_1)_n \mid\mid \langle p_2 \rangle_m = (p_1 \mid\mid p_2)_n$, if $n < m$
	$\langle p_1 \rangle_n \mid\mid (p_2)_m = \langle p_1 \mid\mid p_2 \rangle_n$, if $n < m$
	$\downharpoonleft \pi_1 \mid\mid \downharpoonleft \pi_2 = \downharpoonleft (\pi_1 \mid\mid \pi_2)$
$p \mid\mid p : p \times p \rightharpoonup p$	Intersection of patterns; standard unification-style algorithm
$v \mid\mid_v p : v \times p \rightharpoonup v$	Match v against p; standard unification-style algorithm

Fig. 1. Network Calculus metafunctions.

The function *inject* transforms events from the outside world before adding them to a configuration's event queue. Incoming messages are marked as originating from one layer down and incoming routing table updates are similarly marked with *lift* before they are aggregated with the routes of the network itself:

$$inject : \alpha \times C \to C$$

$$inject\ m\ [\overline{\alpha}\ \overline{\pi}\ \overline{A}] = [\overline{\alpha} \downharpoonleft m \quad \overline{\pi} \quad \overline{A}] \qquad \text{(relay-in)}$$

$$inject\ \overline{\pi}'\ [\overline{\alpha}\ \overline{\pi}\ \overline{A}] = [\overline{\alpha}\ \overline{\pi}_{total} \quad \overline{\pi}_{new} \quad \overline{A}] \qquad \text{(routes-in)}$$

$$\text{where } \overline{\pi}_{new} = lift(\overline{\pi}') \text{ and } \overline{\pi}_{total} = \overline{table(A)}\ \overline{\pi}_{new}$$

A dispatched event results in an enqueued action, which may trigger events within the local configuration, actions for the *containing* network, or both:

$$\overline{a}_0 \lhd [\overline{\alpha} \quad \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : \langle v \rangle \overline{a} \lhd B)\overline{A}] \longrightarrow \overline{a}_0 \lhd [\overline{\alpha}\langle v \rangle\ \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : \overline{a} \lhd B)\overline{A}] \qquad \text{(send)}$$

$$\overline{a}_0 \lhd [\overline{\alpha} \quad \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : (v) \overline{a} \lhd B)\overline{A}] \longrightarrow \overline{a}_0 \lhd [\overline{\alpha}(v)\ \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : \overline{a} \lhd B)\overline{A}] \qquad \text{(feedback)}$$

$$\overline{a}_0 \lhd [\overline{\alpha} \quad \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : \downharpoonleft m \overline{a} \lhd B)\overline{A}] \longrightarrow \overline{a}_0 m \lhd [\overline{\alpha} \quad \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : \overline{a} \lhd B)\overline{A}] \qquad \text{(relay-out)}$$

$$\overline{a}_0 \lhd [\overline{\alpha} \quad \overline{\pi}_1\ \overline{A_Q}(\overline{\pi} : \ \overline{\pi}' \overline{a} \lhd B)\overline{A}] \longrightarrow \overline{a}_0\, drop(\overline{\pi}'') \lhd [\overline{\alpha}(\overline{\pi}''\overline{\pi}_1)\ \overline{\pi}_1\ \overline{A_Q}(\overline{\pi}' : \overline{a} \lhd B)\overline{A}]$$

$$\text{where } \overline{\pi}'' = \overline{table(A_Q)}\ \overline{\pi}'\ \overline{table(A)} \qquad \text{(routes-out)}$$

The rule for spawning new actors is complex. Its essence is as follows:

$$\overline{a}_0 \lhd [\overline{\alpha}\ \overline{\pi}_1 \overline{A_Q}(\overline{\pi} : A_{new} \overline{a} \lhd B)\overline{A}] \longrightarrow \overline{a}_0 \lhd [\overline{\alpha}\ \overline{\pi}_1 \overline{A_Q}(\overline{\pi} : \overline{a} \lhd B)\overline{A} A_{new}] \quad \text{(spawn, draft)}$$

The spawned actor A_{new} is lifted out of the spawning actor's action queue and placed at the end of the containing configuration's actor list. This first draft elides a critical detail, however. Since newly-spawned actors may arrive complete with non-empty subscription sets, the subscriptions must be incorporated into the configuration's routing tables and propagated to other interested parties:

$$\begin{aligned} \ldots \longrightarrow \overline{a}_0\, \mathit{drop}(\overline{\pi}') \lhd [\overline{\alpha}(\overline{\pi}'\overline{\pi}_1)\ \ \overline{\pi}_1\ \ \overline{A_Q}(\overline{\pi} : \overline{a} \lhd B)\overline{A}\ A_{new}] \\ \text{where } \overline{\pi}' = \overline{\mathit{table}(A_Q)}\ \overline{\pi}\ \overline{\mathit{table}(A)}\ \mathit{table}(A_{new}) \end{aligned} \qquad \text{(spawn)}$$

The *entire* subscription table is sent to actors; if an actor needs the difference between the old and the new table, it must perform the computation on its own.

Finally, a network may step if a contained actor state can step:

$$\frac{\Sigma_Q \longrightarrow \Sigma'}{\overline{a}_0 \lhd [\cdot\ \overline{\pi}_1\ \overline{A_I}(\overline{\pi} : \Sigma_Q)\overline{A_Q}] \longrightarrow \overline{a}_0 \lhd [\cdot\ \overline{\pi}_1\ \overline{A_Q}\ \overline{A_I}(\overline{\pi} : \Sigma')]} \qquad \text{(schedule)}$$

This rule allows variations in scheduling. As written, the rule preserves deterministic stepping, picking the leftmost non-inert actor, and it rotates the queue of contained actors, giving each a chance to take a step.

NC satisfies the same basic correctness theorems as our actor calculus. First, the communication layer never fails. Second, the calculus remains deterministic.

Theorem 3 (Soundness). *If interp_0 is total, a behavior B is either inert or there exists some Σ' such that $\cdot \lhd B \longrightarrow \Sigma'$.*

Proof (Sketch). We employ the same Wright/Felleisen technique as for theorem 1, with a slight modification embodied in the progress lemma below.

Definition 1 (Height). *Let the* height *of an actor be defined as follows:*

$$\begin{aligned} \mathit{height} &: A \to \mathbb{N} \\ \mathit{height}(\overline{\pi} : \overline{a} \lhd f; u) &= 0 \\ \mathit{height}(\overline{\pi} : \overline{a} \lhd [\overline{\alpha}\, \overline{\pi}_1\ \overline{A}]) &= 1 + \mathit{max}(\overline{\mathit{height}\ A}) \end{aligned}$$

Let the height of a configuration C be $\mathit{height}(\cdot : \cdot \lhd C)$.

Lemma 1 (Progress). *If interp_0 is total, for all $\overline{a} \lhd C$ and $H \in \mathbb{N}$ with $\mathit{height}(C) \leq H$, C is either inert or there exists some Σ' such that $\overline{a} \lhd C \longrightarrow \Sigma'$.*

Proof (Sketch). By nested induction on the height bound and structure of C. □

Theorem 4 (Deterministic Evaluation). *For any actor state Σ there exists at most one Σ' such that $\Sigma \longrightarrow \Sigma'$ (modulo systematic renaming).*

Proof (Sketch). The proof shows that, due to the restrictions on the scheduling rule, the reduction system cannot create non-trivial diamonds. □

We modeled NC with Redex [10] and Coq [20]; testing the theorems in the former and proving them in the latter.[6]

[6] Models and proofs available at `http://www.ccs.neu.edu/home/tonyg/esop2014/`.

An Interpretation NC comes with several novelties, including concepts such as routing events, subscription, and connection. Together these concepts help address a number of programming problems:

Starting up services Assembling service components into a complete application involves determining a suitable startup order. Otherwise a service may attempt to access another service before the latter is initialised. Routing events solve this problem in a natural fashion. Once a service is ready, it subscribes to incoming requests via $(service, \star)_0$ and therefore its clients can notice it via subscriptions to $\langle service, \star\rangle_1$.

Session management A connection is a relationship between two communicating stateful parties. If some peer A subscribes to $\langle A, c, \star\rangle_0$ (for connection identifier c) and $(B, c, \star)_1$ while B subscribes to $(A, c, \star)_1$ and $\langle B, c, \star\rangle_0$, they not only construct two unidirectional streams, but also each observes the presence of the other. During their conversation, if A receives a routing event in which $\langle B, c, \star\rangle_0$ is *absent*, it knows that B disconnected or faulted and that it may now release any state associated with the connection.

Demultiplexing A network automatically demultiplexes incoming events via subscription-based message filtering. Imagine an NC program that implements an SSH server and uses an SSH-styled protocol. Each SSH packet carries a type identifier number. If each packet type handler subscribes with a pattern identifying a specific type number, e.g. $(ssh, 21, \star)_0$, and each actor responsible for dispatching incoming packets subscribes to $\langle ssh, \star, \star\rangle_1$, the dispatcher can use the resulting routing events to decide whether an "unhandled packet type" error response to an incoming packet is required.

Demand tracking By keeping track of active service instances and monitoring client connections via routing events, "management" actors can match supply to demand for a service, spawning new service instances as clients appear.

VPNs With layering comes a need for coordinating actors, not just direct peers, but also those communicating across levels of containment. By tunneling encoded routing events as messages to remote parties, subscriptions can be propagated between subnets; the relaying actor becomes a proxy for remote peers. This approach is analogous to the topology notifications in distance-vector routing protocols; it yields a form of "virtual private network."

Design Variations Like λ-calculus, NC is a flexible system that can easily serve as the basis for variations and extensions.

Non-determinism While NC is intrinsically concurrent, connecting event-driven and message-exchanging actors, it remains deterministic. Its design carefully ensures that the addition of networks to a deterministic base language yields a deterministic result. Real-world communicating systems are often non-deterministic, however. There are two obvious ways to introduce forms of non-determinism that allow the calculus to exhibit parallelism and racing. First, we can loosen the quiescence and inertness restrictions on the reduction rules. Doing so introduces new interleavings that make the system truly parallel. Second, we can weaken the network's guarantee of delivering messages in order or at all. For out-of-order delivery, the dispatch rule can be modified to select arbitrarily from the queue. For packet loss, the system needs a new rule for discarding messages from the queue. This form of non-determinism primitively reflects the uncertainty that comes with actors relaying messages across layers.

Routing Since routing events do not distinguish entire networks from atomic actors, actors cannot tell the two apart. It is therefore possible to introduce new types of network with the same interface but different internal routing and delivery rules. For example, altering the dispatch rule for message events to select only the *first* actor matching the message, instead of all matching actors, gives "anycast" routing [24]. If, in addition, unroutable events are retained in the event queue until a matching subscription is created, the network behaves as a "message queue" in the terminology of messaging middleware [9].

Furthermore, protocol-specific routing optimizations can be applied to individual layers without breaking encapsulation. For example, IP datagrams are routed on target IP address alone; an IP-specific layer could restrict patterns to permit matching only on target IP address, enabling traditional routing table implementation techniques. In general, each network instance can enforce its own message formats and protocols, for which a session type system [17] is likely to provide the matching static checking.

Fairness NC does not guarantee fairness. If an atomic actor constantly sends itself events, it can starve its siblings. To avoid such starvations, the network could buffer events for atomic actors or rotate the actor queue as part of every action-interpretation step.

Faults and supervision While the interpretation rule also assumes totality, a practical variant of NC can easily handle crashes. If $interp_0$ can return some *exception* token indicating failure, the rule

$$\frac{interp_0\ f\ (\alpha \mid\mid \overline{\pi})\ u = exception}{\overline{\pi} : \cdot \triangleleft f; u \xrightarrow{\alpha} \overline{\pi} : (\cdot) \triangleleft \cdot ; \cdot} \quad \alpha \mid\mid \overline{\pi} \text{ is defined}$$

causes a crashing actor to retract its subscriptions. If a "supervisor" actor [8] exists, it may then deploy matching recovery strategies as failures are detected.

5 Programming with the Network Calculus

Network Calculus, like λ-calculus, is too spare for programming. To make such an exercise reasonably convenient, we assume a purely functional base language extended with a conventional pattern-matching facility. We choose to model atomic behaviors $f; u$ using functions in this base language, meaning that $interp_0\ f\ \alpha\ u = f\ \alpha\ u$. This assumption also means that events α and actions a are data structures in the base language.

To illustrate NC, we implement the chat room of section 3. **Bold** identifiers denote NC terms, `monospace` literal atoms, and *italics* base language concepts.

The chat service is structured as a single network. Contained actors communicate with each other using a chat-network-specific protocol, namely the exchange of $\langle$`chat`, *username*, *text*$\rangle$ messages. Each such message conveys the information that *username* said *text*.

The chat users necessarily exist outside the service itself. Instead of regarding users and their `telnet` connections as meta-entities, we take advantage of the layered structure of NC. While our chat network communicates internally with a chat-specific protocol, we *simulate* the external world as if it were another network layer below the chat network. Actors receive and send messages on both the internal chat network and the simulated network that connects the entire service to the outside world. Ordinary messages are delivered to siblings within the chat network, while arrow-prefixed messages $\rfloor m$ are delivered to the outside world, where the users are. Figure 2 shows the layering.

Our `telnet`-like protocol rests on four message types: $\langle \texttt{connect}, \mathit{username} \rangle$, $\langle \texttt{disconnect}, \mathit{username} \rangle$, $\langle \texttt{input}, \mathit{username}, \mathit{line} \rangle$ and $\langle \texttt{output}, \mathit{username}, \mathit{line} \rangle$. The actors in our chat network are then responsible for (1) interpreting these messages and transforming them into messages for their direct peers on the inner, chat-specific network, and (2) vice versa.

The service's starting configuration **room** both creates the chat network and spawns within it the single stateless actor **acceptor**, which responds to `connect` messages received from the outer network:

$$\begin{aligned} \textbf{room} &= \cdot : \cdot \lhd [\cdot \;\; \cdot \;\; (\cdot : \textbf{acceptor} \lhd \cdot \,; \cdot)] \\ \textbf{acceptor} &= \cdot : \rfloor(\texttt{connect}, \star)_0 \lhd \mathit{acceptor}; \cdot \end{aligned}$$

The $\rfloor$ prefix on the acceptor's subscription indicates that it pertains to the network containing the whole service, shaded in figure 2, rather than the chat-specific inner network. The acceptor wishes to receive `connect` messages from the outside world, but takes action locally in response.

The base-language function *acceptor* implements **acceptor**'s behavior. When it receives a `connect` message, it spawns a relay actor responsible for managing communication with the newly-arrived user:

$$\begin{aligned} \mathit{acceptor}\ \ \rfloor\langle \texttt{connect}, \mathit{user} \rangle\ \mathit{state} &= (\textbf{relay}\ \mathit{user}, \mathit{state}) \\ \textbf{relay}\ \mathit{user} &= \cdot : (\rfloor(\texttt{input}, \mathit{user}, \star)_0 \\ &\qquad\ \ \rfloor\langle \texttt{output}, \mathit{user}, \star \rangle_0 \\ &\qquad\ \ \rfloor(\texttt{disconnect}, \mathit{user})_0 \\ &\qquad\ \ \langle \texttt{chat}, \mathit{user}, \star \rangle_0 \\ &\qquad\ \ (\texttt{chat}, \star, \star)_1) \lhd \mathit{relay}; (\mathit{user}, \{\}) \end{aligned}$$

The **relay** actor advertises subscriptions for `telnet`-like input, output and disconnection events taking place in the outside world, and advertises its intent

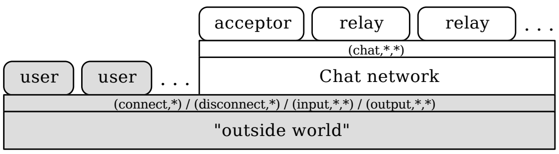


Fig. 2. Chat service layering. Shaded regions are implicit, not part of the program.

to send chat messages into the inner network on behalf of the connected user. All these subscriptions are marked with a subscript 0, because **relay** is only interested in receiving these messages, and is *not* interested in receiving related routing event notifications. In contrast, its final subscription, $(\texttt{chat}, \star, \star)_1$, has a subscript of 1, indicating interest not only in receiving chat messages from the inner network, but also in hearing about related changes to the routing table.

As its peers come and go, their $\langle \texttt{chat}, \mathit{user}, \star\rangle_0$ subscriptions *match* the $(\texttt{chat}, \star, \star)_1$ subscription and are delivered to **relay** as routing events. The actor thus uses information about the routing table to inform the remote user of the arrival and departure of other users. In order to do so, it maintains in its actor state not only its own name but also the set of peers it has seen so far; initially, the empty set $\{\}$.

The base-language function *relay* handles both routing and message events:

$$\begin{aligned}
\mathit{relay}\ \ \rfloor\langle \texttt{input}, \mathit{user}, \mathit{line}\rangle\ (\mathit{user}, \mathit{peers}) &= (\langle \texttt{chat}, \mathit{user}, \mathit{line}\rangle, (\mathit{user}, \mathit{peers}))\\
\mathit{relay}\ \ \rfloor\langle \texttt{disconnect}, \mathit{user}\rangle\ (\mathit{user}, \mathit{peers}) &= (\overline{\pi}', \texttt{nil}) \quad \text{where } \overline{\pi}' = \cdot\\
\mathit{relay}\ \langle \texttt{chat}, \mathit{who}, \mathit{line}\rangle\ (\mathit{user}, \mathit{peers}) &= \\
(\rfloor\langle \texttt{output}, \mathit{user}, \mathit{who} \mathbin{+\!\!+} \text{`` says ''} &\mathbin{+\!\!+} \mathit{line}\rangle, (\mathit{user}, \mathit{peers}))\\
\mathit{relay}\ \overline{\pi}\ (\mathit{user}, \mathit{peers}) &= (\mathit{arrvls} \mathbin{+\!\!+} \mathit{dprt}, (\mathit{user}, \mathit{peers}'))\\
\text{where}\quad \mathit{peers}' &= \{u \mid \langle \texttt{chat}, u, \star\rangle_0 \in \overline{\pi}\}\\
\mathit{arrvls} &= [\ \rfloor\langle \texttt{output}, \mathit{user}, u \mathbin{+\!\!+} \text{`` arrived''}\rangle \quad \mid u \in \mathit{peers}' - \mathit{peers}\ ]\\
\mathit{dprt} &= [\ \rfloor\langle \texttt{output}, \mathit{user}, u \mathbin{+\!\!+} \text{`` departed''}\rangle \mid u \in \mathit{peers} - \mathit{peers}'\ ] \quad (\dagger)
\end{aligned}$$

Text arriving from the user via the remote network is relayed to peers in the chat network. Next, a disconnection notice from the outside world translates into withdrawal of all the relay's subscriptions. Messages from peers are relayed to the user via `output` messages on the outer network. Finally, when a routing event arrives, *relay* computes routing table differences and announces corresponding arrivals and departures to its user.

The subscription withdrawals triggered by `disconnect` events cause routing events to be delivered to other relays. Because subscriptions are being withdrawn, the routing table has shrunk, and so $(\mathit{peers} - \mathit{peers}')$ on line $(\dagger)$ is nonempty, resulting in a "departed" notification being sent to the remaining users.

With the model in place, we can now simulate communication and computation using the *inject* metafunction from section 4. For example, to simulate the connection of user `A`, reduce the configuration state

$$\cdot \triangleleft (\mathit{inject}\ \langle \texttt{connect}, \texttt{A}\rangle\ \mathbf{room})$$

to $\overline{a} \triangleleft C_I'$. The actions $\overline{a}$ include `output` messages for connected users, and C_I' is the final state of the server, waiting for the next event from the outside world.

In this way, *inject* and the resulting $\overline{a}$ provide an I/O interface between an NC program and its context. Our layered structure cleanly accounts for "real I/O" performed by a group of actors in a way that is impossible in a non-layered actor model, lacking any facility for distinguishing actions intended for sibling actors from actions intended for entities outside the actor configuration.

6 Implementing the Network Calculus

Marketplace is a Racket-based [12] implementation of NC. Event handlers are Racket functions, and data structures represent events and actions. Marketplace actor behaviors are also plain Racket functions, meaning $interp_0$ becomes `apply`. In turn, Marketplace's networks are ordinary actors. A second prototype, Marketplace/JS, uses Javascript as the base language and runs in the browser.

To connect to the outside world, Marketplace provides a *ground* network [19]. Its subscriptions are interpreted as subscriptions to Racket's I/O events. It observes the routing table and creates corresponding Racket event descriptors. For example, a Marketplace program may subscribe to a timer or a TCP socket.

Marketplace implements one of the variants of NC discussed in section 4. The Marketplace scheduler is fair. Exceptions thrown by Racket code are translated into failures of actors. Support libraries assist with the manipulation of subscriptions and the interpretation of routing events.

We have written a chat server in Marketplace, comparing it with Python, Haskell and Erlang implementations. Much socket- and state-management is automatic, a consequence of our routing events. Our Python and Haskell implementations initially came with subtle flaws in handling simultaneous disconnections; doing so corrupted shared state in the server. Marketplace avoids such problems *by construction*, with no shared state but the routing table, and no in-place mutation at all.

We have also implemented two major applications to explore Marketplace's potential: a DNS system and an SSH protocol implementation and server.[7]

Our Marketplace **DNS service** is a two-layered network system. While the bottom layer speaks UDP, the upper layer implements a DNS protocol. Relay actors encode and decode DNS packets as they traverse the UDP/DNS layer boundary. Within the DNS layer, actors cooperate to enact the DNS protocol for iteratively discovering the answers to incoming DNS questions. Questions are processed concurrently, with one actor allocated to each DNS inquiry. The system uses broadcasting to keep the internal DNS cache database up-to-date. The cache management actor subscribes to a wildcard so that it can eavesdrop on actors as they communicate DNS answers to each other; it populates the cache based on what it hears.

Our Marketplace **SSH server** consists of three network layers; see figure 3. Its organization directly matches the specification of the protocol [33]. Each new connection results in new Session and Application layer instances. Relay actors receive encrypted TCP data from the ground layer, decrypting and parsing it before sending the results into the session-specific layer. Packet-handler actors in that layer enact the SSH protocol, relaying application data packets to the innermost, application-specific layer. If any actor within the session layer exits unexpectedly, a "watchdog" supervisory actor notices via routing events and disconnects the session. Nesting of layers separates groups of related actors by

[7] All code is available via `http://www.ccs.neu.edu/home/tonyg/esop2014/`.

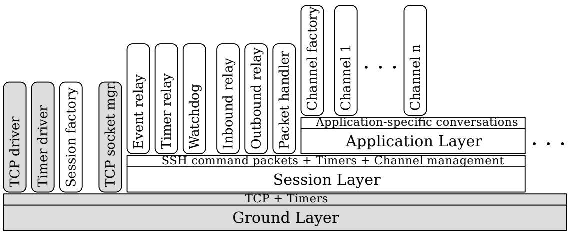


Fig. 3. Layered structure of the SSH implementation

clearly defining the available channels for communication between groups. Each network also provides a crisp boundary for the resources under its control.

Our prototype Marketplace implementations take a simple, unoptimized approach to routing. Nevertheless, the performance of our DNS resolver is adequate; it has been quietly serving web browsers in our lab for the past year.

7 Related Work

On the theoretical level, our work on NC extends previous work on event-driven systems [11], and invites comparison with process calculi and actor-based models of concurrency. On the practical level, our Marketplace language is comparable to actor-inspired languages and their libraries, especially Erlang, Scala, E and AmbientTalk.

In general, most related work concerns point-to-point communication between named entities within a single layer, dealing with broadcasting and layered architectures as derived concepts. In contrast, NC eschews names, treats broadcasting as fundamental, and adds novel routing events. The latter solve many problems: startup ordering, session lifetimes, failures, supervisors, etc. Lacking routing events completely, related systems address these problems on an ad-hoc basis, if at all, rather than as consequences of a unifying mechanism.

The Conversation Calculus Spiritually closest to our work is the Conversation Calculus [3,30], based on π-calculus. Its *conversational contexts* scope multiparty interactions. Named contexts nest hierarchically, forming a tree. Processes running within a context may communicate with others in the same context and processes running in their context's immediate container. Contexts on distinct tree branches may share a name and thus connect transparently through hyperlinks. The Conversation Calculus also provides a Lisp-style `throw` facility that aborts to the closest `catch` clause. This mechanism enables supervisor-like recovery strategies for exceptions.

Although Conversational and NC serve different goals—the former is a calculus of services while the latter is a language design guideline—the two are strikingly similar. Like a network, a conversational context has both a spatial meaning as a location for computation and a behavioral meaning as a delimiter for a session or protocol instance. Both calculi permit communication *within* their respective kinds of boundary as well as *across* them.

The two calculi starkly differ in three aspects. First, NC cannot *transparently* link subnets into logical overlay networks because its actors are nameless. Instead, inter-subnet routing has to be implemented in an explicit manner, based on NC's routing events. Proxy actors tunnel events and actions across links between subnets; once such a link is established, actors may ignore the actual route. Any implementation of Conversation Calculus must realize just such explicit routing; NC can provide the same expressiveness as a library feature.

Second, Conversation Calculus lacks routing events and does not automatically signal peers when conversations come to an end—normally or through failure. Normal termination in Conversation Calculus is a matter of convention, while exceptions signal failure to containing contexts but *not* to remote participants in the conversational context. In contrast, Network Calculus's routing events signal failure to all interested parties transparently.

Finally, our implementation experiences with Marketplace suggest that mapping context names to "wire level" identifiers poses a steep obstacle for a similar effort for Conversation Calculus. After all, different parts of the system are going to be written in different base languages. With the explicit demultiplexing in Network Calculus, managing a heterogeneous system poses no problems.

Actors One major family of Actor models is due to Agha and colleagues [1,4,29].

Varela and Agha's variation [29] groups actors into hierarchical *casts* via *director* actors, which control some aspects of communication between their casts and other actors. If multicast is desired, it must be explicitly implemented by a director. While casts and directors have some semblance to the layered Network Calculus, the two differ in many aspects. Our system's use of pub/sub automatically provides multicast without forcing all members of a layer to use the same conversational pattern. Directors are computationally active, but our networks are not. In their place, Network Calculus employs *relay actors* that connect adjacent layers. Finally, Varela and Agha's system lacks routing events and thus cannot deal with failures easily. They propose mobile *messenger* actors for localizing failure instead.

In Callsen and Agha's ActorSpace [4] actors join and leave *actorspaces.* Each actorspace provides a scoping mechanism for pattern-based multicast and anycast message delivery. Besides communication via actorspace, a separate mechanism exists to let actors address each other directly. In contrast, our system performs all communication with subscription-based routing and treats networks as specialized actors, enforcing abstraction boundaries and making it impossible to distinguish between a single actor or an entire network providing some service. Actors may join multiple actorspaces, whereas Network Calculus actors may only inhabit a single network, reflecting physical and logical layering of networks and giving an account of locality. In our system, actors join multiple networks by spawning proxy actors, which tunnel events and actions through intervening layered networks. Finally, ActorSpace does not specify a failure model, whereas Network Calculus signals failure with routing events.

All actor models lack an explicit interface to the outside world. I/O remains a brute-force side-effect instead of a messaging mechanism. Our functional approach to messaging and recursive layers empowers us to treat this question as an implementation decision.

Mobile Ambients Cardelli and Gordon [5] describe the *Mobile Ambient Calculus.* An *ambient* is a nestable grouping of processes, an "administrative domain" within which computation and communication occur.

At first glance, the two pieces of work are duals. While Network Calculus focuses on routing data between domains, from which code mobility can be derived, Mobile Ambients derives message routing from a primitive notion of process mobility. By restricting ourselves to transporting *data* rather than *code* from place to place, we avoid a large class of mobility-related complication and closely reflect real networks, which transport only first-order data. Moving higher-order data (functions, objects) happens via encodings. Furthermore, mobility of code is inherently point-to-point, and the π-calculus-like names attached to ambients reflect this fact. Our pattern-based routing is a natural fit for a more general class of conversational patterns in which duplication of messages is desired.

Mobile Ambients can directly express *locks*, trading broadcast communication for the ability to express guaranteed-two-party atomic protocols. Network Calculus comes without such locks and guarantees, because communications are always broadcast even if they are intended to be two-party conversations. This seeming weakness is a reflection of our desire to align Network Calculus with the abilities of real networks, which likewise have no means of expressing atomic transfer of ownership. Hence, programs in Network Calculus must, like Actors, implement distributed locking algorithms explicitly.

Process Calculi Fournet and Gonthier's *Distributed Join Calculus* [13] arranges processes in a tree of locations, with automatic mobility and communication between them; Network Calculus manages such nonlocal interaction explicitly. Similarly, neither first- nor higher-order π-calculi [26] represent layered or nested process groups; the spatial arrangement of their processes remains implicit.

Middleware A comparison with publish/subscribe brokers [9] supplies an additional perspective. Essentially, a network corresponds to a broker: the routing table of a network is the subscription table of a broker; the network buffers are broker "queues;" characteristic protocols are used for communication between parties connected to a broker; etc. In short, Network Calculus can be viewed as the first formal semantics of brokers.

Erlang/OTP The closest relative to Marketplace is Erlang/OTP [2,8]. Both support isolated "shared-nothing" message-passing processes; crash reporting in the form of explicit events to interested parties; and supervisory processes. Erlang's `gen_server` interface corresponds closely to our interp_0 signature.

Marketplace differs from Erlang in its use of: broadcasting in lieu of point-to-point communication; abstract topics to name services versus Erlang's use of process IDs; nesting to demultiplex conversations versus explicit demultiplexing; and routing events versus exit signals and process monitors.

Many of the OTP design patterns are linguistic constructs in Marketplace. For example, debugging and tracing of subsystems requires explicit handling of "debug facilities" and "system messages" by OTP processes, whereas in Marketplace the uniform type of an actor behavior allows tracing actors without changing any code. OTP's global service registries are Marketplace's built-in routing tables. Each Erlang application is responsible for solving its startup ordering problem, whereas Marketplace applications can use routing events to find the required topological dependency ordering implicitly.

Scala Several Scala [14] libraries support Actor-style programming. Most of them implement Erlang-style actor supervision. Notable among the implementations is Akka, which arranges actors in a tree—spawned actors are considered children of the spawning actor—and uses the tree as the basis of supervision. Akka's tree arrangement does not constrain communication, and Akka does not support routing events. Akka's *multiple* distinct broadcast mechanisms, especially the EventBus, resemble our pub/sub mechanism. As in Erlang, no special support is provided for solving startup ordering problems.

E and AmbientTalk The E programming language provides language-level support for *vats* [21], which like actors, take atomic *turns* at responding to events. Their state persists between turns for fault-tolerance and recovery. In addition to Miller and his colleagues [22,34], work on AmbientTalk [28] continues to explore vats. AmbientTalk adds distributed service discovery, error handling, anycast and multicast within a mobile, ad-hoc network context. In contrast to Marketplace, AmbientTalk lacks layering and does not exploit pub/sub communication for service discovery, failures, error handling, etc.

8 Conclusion

Existing programming languages fail to support layered communication architectures with linguistic constructs. Instead, programmers develop design patterns and support them with frameworks and libraries. The prevalence of these concepts suggests that language designers should consider the inclusion of appropriate programming constructs. In response, our paper presents a novel language idea—the network—in the form of the Network Calculus, building on existing actor-model designs. With the addition of a network construct, language designers can automatically provide services that programmers routinely redevelop. Programmers in turn can internalize idioms from the networking world to simplify their architectures.

The paper explains how to program in this calculus and how to use it as the basis for a language implementation. We have used Marketplace, our implementation of NC, to create and deploy two major systems. Our experience with this prototype suggests that the resulting applications are more modular than comparable systems while providing sufficient performance for daily use.

NC is a malleable design. In its basic form, it is a deterministic concurrency theory. As discussed, it can readily be extended to a parallel and nondeterministic variant and optimised in protocol-specific ways. We expect to explore both the theoretical framework and its implementation.

Acknowledgements This work was supported in part by the DARPA CRASH program and NSF Infrastructure grant CNS-0855140. The authors would like to thank Olin Shivers and Mitch Wand for listening to many rough presentations on this material. In addition the participants of NU PLT's coffee round posed many helpful questions that helped hone this research.

References

1. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A Foundation for Actor Computation. J. Functional Programming 7(1) (1997)
2. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, Royal Institute of Technology, Stockholm (2003)
3. Caires, L., Vieira, H.T.: Analysis of Service Oriented Software Systems with the Conversation Calculus. In: Proc. 7th Int. Conf. on Formal Aspects of Component Software (2010)
4. Callsen, C.J., Agha, G.: Open Heterogeneous Computing in ActorSpace. J. Parallel and Distributed Computing 21(3), 300–289 (1994)
5. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (Jun 2000)
6. Day, J.: Patterns in Network Architecture: A Return to Fundamentals. Prentice Hall (2008)
7. Ene, C., Muntean, T.: A Broadcast-based Calculus for Communicating Systems. In: Proc. of the Workshop on Formal Methods for Parallel Programming (2001)
8. Ericsson(AB): Erlang/OTP Design Principles (2012), `http://www.erlang.org/doc/design_principles/des_princ.html`
9. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
10. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press (2009)
11. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: A Functional I/O System. In: ICFP (2009)
12. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), `http://racket-lang.org/tr1/`
13. Fournet, C., Gonthier, G.: The Join Calculus: a Language for Distributed Mobile Programming. In: Applied Semantics: International Summer School (2000)
14. Haller, P., Odersky, M.: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
15. Heart, F.E., Kahn, R.E., Ornstein, S.M., Crowther, W.R., Walden, D.C.: The interface message processor for the ARPA computer network. In: Proc. Spring Joint Computer Conference (AFIPS '70). pp. 551–567 (May 1970)
16. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proc. 3rd Int. Joint Conf. on Artificial Intelligence. pp. 235–245. Morgan Kaufmann Publishers Inc. (Aug 1973)

17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. Symp. on Principles of Programming Languages. pp. 273–284 (Jan 2008)
18. Li, P., Zdancewic, S.: Combining Events and Threads for Scalable Network Services. In: Proc. Conf. on Programming Language Design and Implementation. pp. 189–199 (2007)
19. Lieberman, H.: Concurrent Object-Oriented Programming in Act 1. In: Yonezawa, A., Tokoro, M. (eds.) Object-Oriented Concurrent Programming. MIT Press (1987)
20. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), `http://coq.inria.fr`, version 8.0
21. Miller, M.S.: Robust composition: Towards a unified approach to access control and concurrency control. PhD thesis, Johns Hopkins University (2006)
22. Miller, M.S., Cutsem, T.V., Tulloh, B.: Distributed Electronic Rights in JavaScript. In: Proc. European Symp. on Programming (2013)
23. Object Management Group: Data Distribution Service for Real-time Systems (Jan 2007)
24. Partridge, C., Mendez, T., Milliken, W.: Host Anycasting Service. RFC 1546 (Informational) (Nov 1993), `http://www.ietf.org/rfc/rfc1546.txt`
25. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Mar 2011)
26. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press (Oct 2003)
27. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
28. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In: Intl. Conf. of the Chilean Society of Computer Science (SCCC). pp. 3–12. IEEE (Nov 2007)
29. Varela, C.A., Agha, G.: A Hierarchical Model for Coordination of Concurrent Activities. In: Proc. Int. Conf. on Coordination Languages and Models (1999)
30. Vieira, H.T., Caires, L., Seco, J.a.C.: The Conversation Calculus: A Model of Service Oriented Computation. In: European Symp. on Programming (2008)
31. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A Note on Distributed Computing. Sun Microsystems Laboratories Technical Report SMLI TR-94-29 (Nov 1994)
32. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115, 38–94 (1992)
33. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Jan 2006)
34. Yoo, S., Killian, C., Kelly, T., Cho, H.K., Plite, S.: Composable Reliability for Asynchronous Systems. In: Proc. USENIX Annual Technical Conference (Jun 2012)
35. Zave, P., Rexford, J.: The geomorphic view of networking: A network model and its uses. In: Proc. of the Middleware for Next Generation Internet Computing Workshop (2012)