Distributed Data Flow Language for Multi-Party Protocols

Krzysztof Ostrowski
Cornell University
Ithaca, NY 14853, USA
krzys@cs.cornell.edu

Ken Birman
Cornell University
Ithaca, NY 14853, USA
ken@cs.cornell.edu

Danny Dolev
Hebrew University
Jerusalem, 91904, Israel
dolev@cs.huji.ac.il

ABSTRACT

This paper presents a new object-oriented (OO) approach to modeling the semantics of distributed multi-party protocols such as leader election, distributed locks or reliable multicast, and a programming language that supports it. The approach extends our *live distributed objects* (LO) model with the new concept of a *distributed flow* (DF), a stream of events that flow concurrently at multiple locations. DFs correspond to local variables, private fields,and method parameters in Java-like languages; they're means by which one stores and communicates state. Protocol instances correspond to Java objects; they consume and output flows; their internal states are encapsulated as internal flows, and their internal logic is represented as operations on flows. Our language provides a new type of concern separation: the semantic structure of protocols is decoupled from implementation details such as construction and maintenance of overlays, trees, and other structures used for scalability. These can be generated by the compiler or at deployment time. This can be done differently in different parts of the network, to match the local environment.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Reliability, Theory

Keywords

Distributed Data Flows, Distributed Multi-Party Protocols, Live Distributed Objects, Monotonic Aggregation

1. INTRODUCTION

The premise of this work is that *distributed multi-party protocols* (DMPs) such as virtual synchrony (VS), two-phase commit (2PC), and Paxos are becoming increasingly important and are used pervasively, and that growing popularity of cloud and edge computing will require that developers be able to design their own DMPs.

Our goal is to provide a simple, expressive DMP definition language that allows developers to express semantics concisely, using high-level constructs, and such that the flow of state and decisions can be readily understood from the code, not obfuscated by internal details such as sending individual network messages from A to B.

Programming DMPs is hard, but it could be simplified with tools that promote a separation of concerns. Developers should be able to specify the semantics and *logical* control flow without having to explicitly handle *physical* aspects, such as failures, timeouts, network topology, and organizing nodes into trees, rings, or other scalable structures. The latter can and should be treated as orthogonal, much as compiler optimizations in C are orthogonal to code semantics. To enable this, we need a set of programming abstractions that are powerful enough to model common DMPs, but leave enough flexibility for the compiler to generate scalable code. This inherent tension between expressiveness and compiler flexibility has been the main factor that shaped our approach and our design decisions, and that distinguishes this work from the existing protocol languages.

Before going further, let's comment on some of the points made earlier. First, we stated that DMPs are getting increasingly important and used pervasively. In the past, DMPs have been used mostly in data centers (DCs), financial institutions or military settings, e.g, to replicate services and data, for load-balancing or fault-tolerance, or to coordinate configuration changes and synchronize access to services [5]. In these senarios, DMPs run among servers in DCs, whereas the larger Web has remained predominantly client-server; home user's machines don't communicate with one-another. In prior work [23], we argued that this trend is bound to change. Home computers, equipped with ever-increasing amounts of memory and multi-core CPUs, are getting faster, whereas web content providers stumble over scalability as their users bases expand. Many types of dynamic, interactive, short-lived content (collaborative work, interactions in virtual worlds) are hard to cache and index and can be hard to scale by adding more servers. It's only natural to off-load servers by pushing data out of data centers, and towards the clients. Technologies based on this idea already exist. In our *live distributed objects* (LO) platform [1], visual elements on an interactive web page – chat windows, video streams, and shared documents – can be individually powered by DMPs; their contents don't reside on servers; they are replicated among the clients in a peer-to-peer fashion. The DMP that runs among the clients ensures that all replicas stay in sync[1]. The creators of Smalltalk [16] used similar approach as a basis of their Croquet [27] platform; 3D objects in their virtual space are replicated with a variant of 2PC . Darkstar [28] and a few other [7] projects also fall into this category. Each leads to a pervasive deployment of the associated DMPs.

PLOS '09 Big Sky, MT, USA

[1] We encourage the reader to watch videos on the *Live Distributed Objects* project website [1] to build intuition behind our approach.

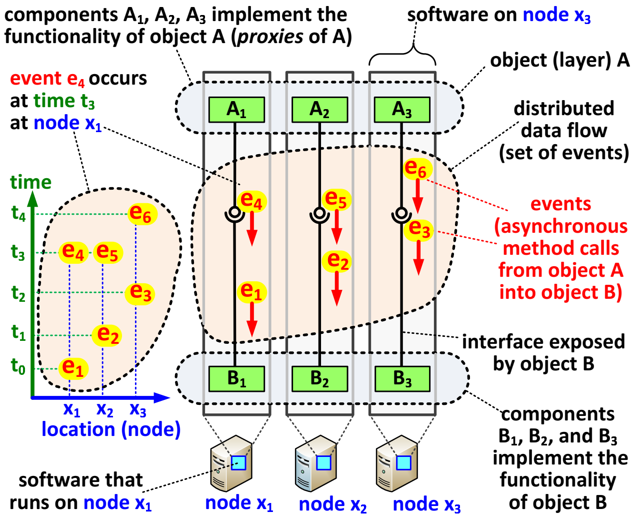
components A_1, A_2, A_3 implement the functionality of object A (*proxies* of A)
software on node x_3
object (layer) A
event e_4 occurs at time t_3 at node x_1
distributed data flow (set of events)
time
t_4
t_3
t_2
t_1
t_0
e_6
e_4
e_5
e_3
e_2
e_1
A_1
A_2
A_3
events (asynchronous method calls from object A into object B)
interface exposed by object B
B_1
B_2
B_3
x_1 x_2 x_3
location (node)
components B_1, B_2, and B_3 implement the functionality of object B
software that runs on node x_1
node x_1
node x_2
node x_3

Figure 1: A *distributed data flow* (DF) is defined as a set of events exchanged between two protocol layers. Each DF is distributed in space (the events can appear on different nodes), and in time (new events flow over time). Each protocol layer involves a set of components (such as A_1, A_2, A_3) across a set of nodes; each group of such components is called a *live distributed object* (LO). Elements of the group are called *proxies*. Here, layer A is an LO that consists of three proxies A_1, A_2, and A_3. One can think of each LO (each protocol layer) as transforming some set of input *flows* into some set of *output* flows (for more detail, consult [25]).

The second premise of our work is that programmers will want to create their own DMPs. Distributed computing forces them to choose between reliability, scalability, performance and persistence, and different applications require a different balance. For example, the version of reliable multicast DMP used for database replication in a financial institution would require a consensus semantics, but would not need to scale to thousands of nodes, whereas the variant of reliable multicast used to synchronize players in an online multiplayer game (MMORPG), or clients watching a streaming movie, would require excellent scalability at the cost of weaker semantics. In other work [23], we pointed out that even for a seemingly simple task such as collaborative editing, there exists a surprising variety of different approaches that rely on different ways of locking, reconciliation, and flavors of multicast, often fine-tuned to the particular application domain. The analogy to Java and .NET collections seems appropriate: even though many applications don't need custom collections and can be built using the small set of built-in ones, such as lists, arrays, and hash tables, those who build high-performance or scalable systems often design their own custom collections optimized for their specific applications. Compared to collections, DMPs and their tradeoffs are even more complex and diverse.

Designing DMPs in Java-like languages is hard; popular toolkits like Ensemble [13], Spread [2], and Appia [22] have 25,000+ lines of code. Systems such as MACE [17] can reduce the programming burden, but programmers still have to reason at the level of states, transitions, and network messages sent between pairs of nodes; this may be easy for loosely-coupled systems such as distributed hash tables (DHTs), but it can be hard for DMPs. One way to simplify the process is by composing pre-existing reusable protocol layers in DMP composition toolkits such as Ensemble, Spread, Appia, or BAST [11]. This approach is convenient, but there are limitations. First, to achieve a high degree of flexibility, one needs a large number of thin and simple protocol layers: Ensemble has 50. Even then, flexibility is limited, for only certain combinations of layers make sense. Flexibility in these systems generally amounts to including (or not) certain functional layers, e.g., ordering. To use a *different* ordering scheme, one generally has to write a custom layer in Java, which requires familiarity with the given DMP composition toolkit and its API. Finally, while the DMP toolkits can separate functional layers from one-another, their functionality is often tightly coupled to implementation; e.g., a layer that handles recovery, ordering, or stability may be hard-wired to aggregate its information in a particular manner, such as by using a leader or *all-to-all* communication, and may be unable to easily switch to gossip, trees, or token rings. The latter is also a weakness of MACE and other systems that force programmers to work at the level of state transitions and network messages; code that builds distributed structures gets intermingled with and inseparable from core semantics and logical control flow.

In this paper, we advocate a different approach: we propose a few simple generic abstractions that can be easily composed to express semantics as diverse as distributed agreement and leader election, that can be stacked hierarchically to express scalable hierarchical protocols, and that can themselves be implemented in a variety of ways, such as by using token rings, trees, gossip, or IP multicast. Protocols in our language are compact and easy to reason about, yet at the same time they leave the runtime a high degree of flexibility in mapping language constructs to an executable code.

Our approach is based on four fundamental concepts: *distributed data flows* (DFs) as a way to model state and semantics, distributed *monotonic aggregation* (MA) as a means of coordinating DMP participants, the use of *set arithmetics* (SA) to model batched processing, and the use of *recursion* in the language to express hierarchical distributed structures maintained for scalability. Due to limited space, this paper discusses only aggregation and batching. The DF concept, and the underlying theory for how strong reliability properties can be expressed through monotonic aggregation, have been discussed in our past work [25], and a brief discussion of recursion can be found in our tech report [24]. The latter includes a language grammar and more protocol examples (omitted here for brevity).

2. LANGUAGE

Each program in our language implements a DMP (for an example of a distributed locking code, see Figure 2). A running *instance* of such program is an instance of a DMP running on a set of nodes across the network; we refer to the latter as a *live distributed object* (LO), or simply an *object* [23]. A single instance of a software component that encapsulates a DMP stack and runs on a single node is called a *proxy*. Each *object* comprises of a *set of proxies*, and spans a portion of the network. Its proxies interact with the application instances on the nodes on which it runs. The object can be thought of as a medium over which those application instances on different nodes communicate with one-another and coordinate their actions.

We also use the term *object* (LO) to refer to individual functional layers within an instance of a complex DMP. For example, a reliable multicast object may consist of an unreliable multicast object composed with a loss recovery object, and the latter may internally use membership and transport objects. In general, the term *object* can refer to any set of software components that run across a set of networked nodes, are functionally related, and expose similar APIs.

The API exposed by each LO proxy is modeled as a set of *event queues* (EQ); each event represents a single method call, a response to such call, or a callback. Each proxy of the LO exposes the same set of EQs. Each EQ transfers a single type of events, in one direction. For example, each proxy of a reliable multicast object would expose two EQs: one for **send** calls, the other for **receive** callbacks.

The set of events that appear on some set of EQs is called a *distributed data flow* (DF); we assume that all these EQs carry events of the same type and in the same direction: into the object (method calls), out of the object (callbacks), or internally in the object (e.g., events stored in *owner* EQs within the proxies of *lock* on Figure 3). Accordingly, one can classify DFs as *input*, *output*, or *internal* with respect to the object. For example, a reliable multicast object would have a single input flow *send* and a single output flow *receive*. The *send* flow would consist of all invocations of the *send* method exposed by instances of the DMP stack. Note that a DF includes calls that may occur at different nodes and at different times; in general, a DF is *distributed* in both of these dimensions (Figure 1).

Within the model introduced above, one can think of each object (each running DMP instance, and each functional layer within such instance) as a *distributed* function that transforms some set of input DFs into a set of output DFs; e.g., an instance of a reliable multicast DMP can be thought of as a distributed function that transforms the *send* input DF into the *receive* DF. We think of this function as *distributed* because the transformation doesn't happen synchronously, and it does not occur at a single location: a single *send* event may trigger a set of *receive* events at other nodes running the DMP. This last factor explains why LO and DF are defined as distributed: they need to be such for us to be able to capture a distributed semantics.

The functional nature of our model motivates the syntax: a DMP is expressed in a way similar to an ordinary function, with its name followed by a list of input DFs in parentheses, a colon, and a list of output DFs; e.g., the locking protocol (*lock* on Figure 2) transforms an input DF named *wants* into an output DF named *holds* (line 01). The types of events in the DFs are denoted similarly to the types of arguments in C; in *lock*, both the input and output DFs are Boolean. Each event flowing into *lock* represents a single request to acquire (when the event is carrying a *true* value) or release the lock (when the even is carrying a *false* value) by the local application instance that has generated the event. The receipt of such event prompts the proxy of *lock* at which the event arrives to coordinate with proxies of *lock* on other machines, to determine the course of action. When a decision is made, proxies of *lock* relay it to their local application instances by issuing event *wants*; again, the *true* value in such event means that the lock is granted, and *false* confirms it's been revoked. The reason why different classes of requests (such as acquiring and releasing locks) are carried by events in the same flow (e.g., *wants*) is that this enables us to express decisions in output flows in a functional manner, as expressions computed over values in input flows.

When compiled, a program in our language yields the executable code of a single proxy; e.g., code on Figure 2 compiles to a structure shown on Figure 3. Each input or output flow is translated into a single EQ within the proxy; these EQs constitute the proxy's API. Each internal flow (declared as in line 02 on Figure 2) is translated into an EQ encapsulated within the proxy; this is used in a manner similar to a local variable in Java (note the EQ *owner* on Figure 3). Program body consists of *dependencies* that express flows in terms of other flows. Each dependency is translated to a code that moves and translates events between EQs (within and across the proxies). For example, the dependency in line 05 (Figure 2) pulls events from EQs *wants*, *owner*, and *id* (*id* is a built-in internal EQ that provides each proxy with its globally unique identifier), applies operators $\wedge$ and $=$, and pushes the result into the local *holds* EQ. Whenever a new value appears in *wants* or in *owner* (*id* is a constant flow), the expression is recalculated and a new event appears in the *holds* EQ.

While similar, our programs differ from Java functions in several respects. First, the code does not execute just once, producing output synchronously from input; rather, values in input flows are continuously fed into the dependencies, and these continuously place

```
01: object lock ( bool wants ) : bool holds { // intput & output flows
02:     int owner; // an internal flow
03:     where (wants) // this determines who runs the code in line 04
04:         owner := stable_elect(id); // an embedded leader election
05:     holds := wants ∧ (owner = id); } // flow dependency
```

Figure 2: Distributed locking in our language: an object named *lock* consumes a Boolean flow $wants$, and generates a Boolean flow $holds$ (line 01). An internal flow $owner$ (line 02) stores the identifier of the node that holds the lock. All nodes that would like to acquire the lock (line 03) fetch identifiers to the embedded object $stable\_elect$ (line 04; for the code consult Figure 4). The lock is held if local id matches that of the leader (line 05). The result of election remains stable until the owner leaves [24].

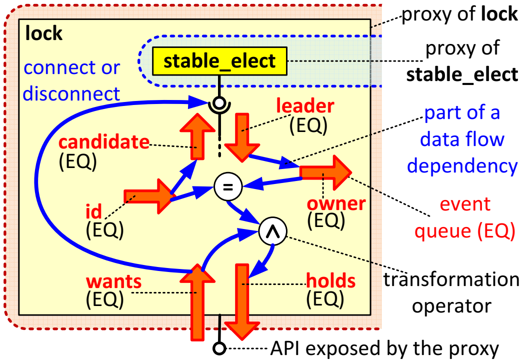

Figure 3: Dependency in line 04 in Figure 2 embeds proxies of $stable\_elect$ in proxies of $lock$, binds $stable\_elect$'s input to flow id, and routes its output into $owner$. Values in $holds$ are generated from those in id, $wants$, and $owner$ (line 05). Values in $wants$ activate or deactivate the connection to $stable\_elect$.

```
01: object stable_elect ( int candidate ) : int leader {
02:     int elected := 0;
03:     where (fresh elected ∧ elected ≤ candidate) // guard
04:         elected := min candidate; // distrib. monotonic aggregation
05:     leader := elected; }
```

Figure 4: A simple version of the leader election protocol. Candidates with identifiers larger than the one of the elected leader (line 03) select among themselves the one with the smallest id (line 04). Result can change only if the leader leaves [24]. Candidates with ids smaller than the leader abstain from election until existing members quit, causing aggregation to reboot [24].

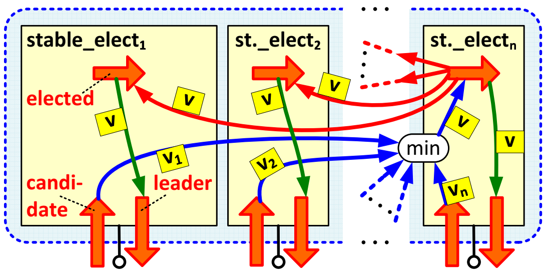

Figure 5: A group of proxies computing aggregation in the protocol from Figure 4. In each aggregation round, sets of values v_i that appear in flow $candidate$ are aggregated into a single value $v = \min_{1 \leq i \leq n} v_i$ (line 04) that emerges at the ring leader node. The result is disseminated to all proxies. Proxies self-organize into a token ring with the help of a built-in *membership service*.

new events in the output flows. Secondly, the order of dependencies is irrelevant; they all execute concurrently. Finally, the execution is not always local; sometimes it is coordinated among proxies.

The universal mechanism for coordinated execution in our model is *monotonic aggregation* (MA), expressed as in line 04 (Figure 4), by applying an operator such as *min*, *max*, *sum*, $\cup$, or $\cap$ (and others) to a single argument. This type of dependency causes values from a given EQ (*candidate* EQ in this case), from group of proxies, to be aggregated using the respective operator (here *min*), and the result disseminated back to all proxies and placed in the target EQ (here, we place the result in *elected* EQs). This is illustrated on Figure 5. By default, aggregation has certain important properties that make it possible to build strong semantics. The MA concept is discussed at length in our past work [25]. Due to limited space, here we limit ourselves to pointing out that MA could be translated to a simple token ring that self-organizes using an external membership service, but it can also be translated into a scalable, hierarchical architecture [25]. MA is a very simple, lightweight and scalable abstraction that can be composed to express a great variety of complex DMPs with strong semantics such as atomic delivery or consensus. Correctness arguments for *lock* and *stable_elect* are given in a tech report [24].

Like most languages, ours also supports a conditional execution. The **where** clause locally activates the embedded code whenever a *true* value flows in the EQ corresponding to the condition in parentheses, and deactivates it when *false* flows in it; e.g., code in line 04 (Figure 2) is active only during times when the last value that was placed in the local *wants* EQ is *true*; otherwise, line 04 is disabled on the particular proxy. For an aggregation, being disabled means that the proxy doesn't contribute values to aggregation; this will be the case whenever (**fresh** *elected* $\wedge$ *elected* $\leq$ *candidate*) evaluates to *false* in line 03 on Figure 4 (expression **fresh** *elected* is *true* when the proxy knows that it has the latest aggregated value of *elected*; in token ring protocols this is easy to determine; for details, see [24]). Unlike in Java or C, our conditional statement does not choose between two execution paths, but rather *expands* or *shrinks* the set of proxies that participate in the given part of distributed computation.

The last important feature used in the *lock* example is the reference to *stable_elect* in line 04 (Figure 2). This type of dependency embeds proxies of the referenced protocol (*stable_elect*) in proxies of the DMP being defined (*lock*), as shown on Figure 3. Each EQ for an input or output DF of the embedded proxy of *stable_elect* is now also an internal EQ in the proxy of *lock*. Events from EQs corresponding to flows passed as arguments (*id*) and results (*owner*) of the assignment are pushed back and forth between EQs to establish communication between the proxies (consult Figure 3). Internals of the embedded proxy of *stable_elect* are obscured to *lock*. Hence, this pattern implements the OO encapsulation principle.

In the examples discussed so far, all flows were carrying simple scalar Boolean and integer values. To express batched processing, we equipped our language with support for set arithmetic similar to that in SETL [26]. The most common use of this feature is to express within a single event a set of identifiers of network messages for which a certain property holds; e.g., in the *stabilize* protocol on Figure 6, each value in the *received* DF represents a set of ids of messages that are being reported as locally received. By encoding a set of identifiers within a single set value, the application using *stabilize* (in this case, the "application" is a reliable multicast object that uses *stabilize* to determine when it can safely deliver its messages) can report a number of application events (message receipts) within a single operation (a single event flow). Furthermore, receiving information in this compressed form, *stabilize* can also process events in parallel: when it calculates a set intersection of values received by different proxies (line 04), *stabilize* can identify multiple network messages as "stable" in a single aggregation round. This is done by intersecting the respective set values (a message is "stable" if it's been received everywhere in the group; the interpretation of *everywhere* in our model has been discussed in our past work [25]). In our simulations [25], token ring aggregation at a few rounds/s made commit and abort decisions for thousands of transactions/s by using set arithmetic. We've also used this approach successfully in our hand-coded high-performance scalable multicast platforms.

```
01: object stabilize ( {int} received ) : {int} stable {
02:     {int} received_by_all := ∅;
03:     where (fresh received_by_all ∧ received_by_all ⊆ received)
04:         received_by_all := ∩ received;
05:     stable := received_by_all; }
```

Figure 6: Code that determines which packets are *stable*, i.e., received by everyone in the system; this computation is an essential component of many reliable multicast protocols.

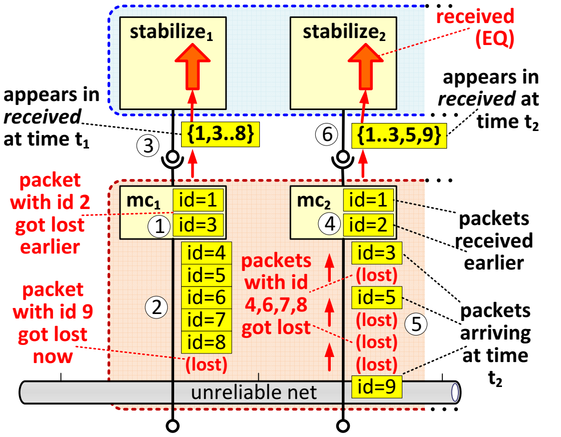

Figure 7: Batched processing with set arithmetic in $stabilize$: (1) before time t_1, node 1 received messages with ids 1 and 3, and is caching them in its local proxy mc_1 of the multicast object; message with $id = 2$ never arrived; (2) at time t_1, a batch of packets with ids from 4 to 8 is received; (3) proxy mc_1 now reports its status to the local proxy of $stabilize$; an event with a single set value $\{1, 3..8\}$ flows at that node at time t_1 and is placed in the $received$ EQ in $stabilize$; (4..6) a similar scenario on node 2 results in $\{1..3, 5, 9\}$ being put into the $received$ EQ.

3. RELATED WORK

Most of the existing protocol-modeling languages are based on the *finite state machine* (FSM) model: every protocol participant (a *proxy* in our terminology) is represented as a finite automaton, with transitions triggered by timeouts, the receipt of network messages, or application requests. A programmer defines all states and transitions, and the compiler translates the high-level FSM specification to executable code, automating aspects such as socket operations, serialization, logging, and verification. MACE [17] and nesC [12] are prominent examples of this approach in the context of loosely-coupled distributed systems. Most of the prior work in this category targeted point-to-point protocols such as TCP (for the list, see [24]).

Besides translation to code, FSM has also been used for program analysis: high-level specifications in Promela [14] and TLA [19] can be translated to FSMs for model checking. TLA is sufficiently expressive to accurately capture strong semantics such as distributed consensus . SOA and WS-* standards for describing peer-to-peer interactions, such as WSCL [3], are also founded on FSM.

Researchers argued [17] that FSM-based languages are natural to work with: the FSM logic resembles a well-written Java code while being more concise. However, MACE-like systems have been used mostly for loosely-coupled systems, such as DHTs or overlays. Expressing DMPs such as reliable multicast or agreement via states, transitions and point-to-point messages can be difficult [6, 13, 15]. Also, as noted earlier, core semantics (making decisions, reconfiguration, and state recovery) is mixed with code that builds distributed structures for dissemination or aggregation. For the sort of concern separation we postulated, a higher-level language is needed.

P2 [20] is a higher-level model: it replaces explicit point-to-point communication with rules in Datalog that create dependencies between local variables at different nodes; point-to-point communication is then generated automatically. This results in compact code, but operating at this level, without tools such as consistent aggregations or membership that are built into our language, it would be hard or impossible to achieve stronger semantics; indeed, P2 has been used primarily with overlays, DHTs, and routing. The same is true for languages based on process calculi; they cannot express strong semantics [10]. In contrast to all these approaches, our language supports aggregation, recursion, batched processing, and essential object-oriented (OO) features, such as encapsulation.

There's been much research on embedding group-like distributed abstractions in higher-level languages such as ML [18] and Java [8]; surveys can be found elsewhere [4, 23]. Unlike our language, these weren't designed to construct protocols, but rather to embed entire existing protocols in strongly typed or OO languages. BAST [11] goes further, in that it supports extensibility by inheritance, but BAST protocols are coded in Java, much as in other DMP toolkits: Spread [2], Ensemble [13], and Appia [22]. The reasons why we prefer a dedicated language have been articulated earlier.

Our DF semantics is functional in spirit; in this sense, our work is inspired by I/O automata (IOA) [21]. However, IOA is a specification language, and doesn't yield executable code. In comparison to IOA, our work is also less focused on individual endpoints and their state, and more on data flows. This creates flexibility that can be exploited to achieve the concern separation we postulated: we can deploy the same protocol over different aggregation, dissemination, batching mechanisms, or differently constructed hierarchies.

Data flows in the sense of asynchronous, massively parallel, pipelined processing, have a long tradition in areas such as VLSI/DBMS, and aggregation has been extensively studied in the context of sensor networks; a discussion of the relevant prior work can be found elsewhere [24], [25]. The key difference is that data flows in those systems are not *distributed* in the same sense as in our model: they are point-to-point event streams, transformations on them are local, and they lack the sort of strong semantics needed to express DMPs.

Many specific solutions we employed have been inspired by prior research: set arithmetics in SETL [26], event-driven computing in SEDA [29] and rule-based computing in Rête [9], to name a few.

4. CONCLUSIONS

We proposed a new type of a programming language for distributed computing that abstracts away low-level details such as point-to-point communication, while retaining sufficient expressiveness to model complex DMPs such as distributed locking, agreement, election, or reliable multicast. Focusing on data flows, their functional dependencies, and distributed constructs such as consistent aggregation, and moving away from endpoint-centric aspects such as states and transitions, allows us to separate semantics from details such as methods of aggregation, construction or hierarchy maintenance. Our distributed flow concept promotes concise code and can facilitate formal reasoning about global system behavior.

5. REFERENCES

[1] Live Distributed Objects. *http://liveobjects.cs.cornell.edu/*.

[2] Y. Amir and J. Stanton. The Spread wide area group communication system. *J. Hopkins Univ. Tech Report*, 1998.

[3] A. Banerji et al. Web Services Conversation Language. *http://www.w3.org/TR/wscl10/*.

[4] J. Briot, R. Guerraoui, and K. Lohr. Concurrency and distribution in object-oriented programming. *CSUR*, 1998.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, 2006.

[6] G. Chockler, I. Keidar, and W. Vitenberg. Group communication specifications: A comprehensive study. *CSUR*, 2001.

[7] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera. Enabling massively multiplayer online gaming applications on a P2P architecture. *ICIA*, 2005.

[8] P. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: abstractions for publish/subscribe interaction. *ECOOP*, 2000.

[9] C. Forgy. On the efficient implementation of production systems. *Ph.D. thesis*, *CMU*, 1979.

[10] R. Fuzzati and U. Nestmann. Much ado about nothing? *http://www.brics.dk/NS/05/3/*, 1995.

[11] B. Garbinato and R. Guerraoui. Using the strategy pattern to compose reliable distributed protocols. *COOTS*, 1997.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *PLDI*, 2003.

[13] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. *TACAS*, 1999.

[14] G. Holzmann. The model checker spin. *TSE*, 1997.

[15] D. Karr. Specification, composition, and automated verification of layered communication protocols. *Ph.D. Thesis*.

[16] A. Kay. The early history of smalltalk. *HOPL*, 1993.

[17] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distr. systems. *PLDI'07*.

[18] C. Krumvieda. Distributed ml: Abstractions for efficient and fault-tolerant prgramming. *Cornell Univ. Tech Report*, 1993.

[19] L. Lamport. The temporal logic of actions. *TOPLAS*, 1994.

[20] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SOSP*, 2005.

[21] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *PODC*, 1987.

[22] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. *ICDCS*, 2001.

[23] K. Ostrowski. *Live Distributed Objects*. Ph.D. Dissertation, Cornell University, 2008. http://hdl.handle.net/1813/10881.

[24] K. Ostrowski, K. Birman, and D. Dolev. Programming Live Distributed Objects with Distributed Data Flows. *Cornell Univ. Tech Report*, 2009. http://hdl.handle.net/1813/12766.

[25] K. Ostrowski, K. Birman, D. Dolev, and C. Sakoda. Implementing reliable event streams in large systems via distributed data flows and recursive delegation. *DEBS*, 2009.

[26] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. Programming with sets: An introduction to setl. 1986.

[27] D. Smith, A. Kay, A. Raab, and D. Reed. Croquet: a collaboration system architecture. *C5*, 2003.

[28] J. Strohm. Managing player awareness in Darkstar. *http://www.projectdarkstar.com*, 2007.

[29] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SOSP*, 2001.