Recursion in Scalable Protocols via Distributed Data Flows

Krzysztof Ostrowski
Google, Inc.
ostrowski@google.com

ABSTRACT

This paper proposes a new approach to representing scalable hierarchical distributed multi-party protocols, and reasoning about their behavior. The established endpoint-to-endpoint message-passing abstraction provides little support for modeling distributed algorithms in hierarchical systems, in which the hierarchy and membership dynamically evolve. This paper explains how with our new *Distributed Data Flow* (DDF) abstraction, hierarchical architecture can be modeled via recursion in the language. This facilitates a more concise code, and it enables automated generation of scalable hierarchical implementations for heterogeneous network environments.

1. INTRODUCTION

Distributed multi-party protocols to coordinate very large groups of machines have been implemented in a hierarchical fashion for decades [1, 2] and there exists much prior research on infrastructure that helps large groups to self-organize [4]. Hierarchy is often used in heterogeneous environments, such as those spanning WANs / LANs because the cost of message passing very often varies between different layers of the network and the availability of mechanisms such as IP multicast varies between local administrative domains. In a hierarchical system, coordination can be gradually achieved, e.g., by a tree of micro-protocols that span small sets of participants.

For example, a system of 9 nodes might self-organize into three 3-node subgroups: $\{A_i\}$, $\{B_i\}$, $\{C_i\}$, where $1 \leq i \leq 3$. The nodes in each group could run local micro-protocols P_A, P_B, P_C, and one selected node from each group (e.g., A_1, B_3, and C_2) could participate in a higher-level micro-protocol Q. If the goal is to achieve agreement, each local micro-protocol P_j, $j \in \{A, B, C\}$ would achieve agreement locally, and the global micro-protocol Q would combine these partial results into a single global decision. Semantics as complex as virtual synchrony can be implemented in this fashion [1, 2]. At least in principle, the architecture has good potential to scale, and each micro-protocol could be implemented differently to take full advantage of the local characteristics of its environment. An example of the latter type of flexibility has been recently demonstrated in the context of collaborative applications [3].

In practice, the potential has never been fully realized. In the prior work just mentioned above, hierarchy is just 2 level deep, and in most existing work, communication patterns at each level are fixed. Few implementations exist. Systems of this sort tend to be complex, and notoriously hard to debug. Recent work [5, 9] has done much to automate many tedious tasks involved in implementing distributed protocols, but it has not changed the fundamental abstraction: the point-to-point exchange of messages between individual participants.

In prior work [6, 8], we argued that point-to-point message passing is too low-level an abstraction for constructing distributed protocols, and we proposed *Distributed Data Flows* (DDF): a new, higher-level language abstraction and formalism for modeling scalable systems. This paper complements that prior work: it focuses specifically on the use of recursion in the language to concisely represent hierarchical behavior.

2. FROM RECURSION TO DATA FLOWS

Consider how we could express a hierarchical protocol via recursion. At a glance, the answer might seem obvious, since the hierarchy itself has a recursive structure. In our example from Section 1, the full set of 9 nodes decomposes into three 3-node groups. Conceptually, it seems plausible that each of the micro-protocols P_A, P_B, P_C, and Q should do the same, and it should be described using the same fragment of code. Instances of this code would operate at different granularity, but otherwise model the same general flavor of computation, such as merging N proposals into a single decision. However, in trying to make this more concrete, we run into problems.

Consider node B_3, which in our example plays a dual role: it participates in a micro-protocol P_B, through which all B_i decide on a single local value, and then feeds this value into the global micro-protocol Q which it runs along with A_1 and C_2. Thus conceptually, on node B_3 the partial results of P_B are being communicated up to Q. Now, suppose B_3 crashes, and two new, fresh machines B_4 and B_5 join the group $\{B_i\}$. Suppose that after a reconfiguration, B_4 assumes the distinguished role that B_3 had played, and starts to participate in protocol Q. Now, it is B_4 that communicates results of P_B up to Q. How should our recursive program deal with that?

First, note that in practice, when we talk about recursion, we expect a degree of encapsulation: the details of the nested computation or data should never be exposed to the "outer" context. Thus, the crash of B_3 and joining of B_4, B_5, should be transparent to instance Q of our recursive program. They should be handled internally by instance P_B of this program.

While one could debate whether this style of encapsulation is really necessary on philosophical grounds, there is a strong practical reason to enforce it. In many distributed protocols, the flow of information from P_B to Q needs to be in a certain sense[1] consistent over time. In our case, if nodes running P_B have agreed upon a certain decision, and if that decision has been communicated to Q, any future decision that P_B reach upon must be consistent with this past report, for otherwise the entire structure falls apart: it is hard to even specify the semantics of Q if its input can change over time[2]. This leads us to the realization that from Q's perspective, there has to

[1]This has been discussed in more precise terms in other work [8].
[2]When formally defining distributed agreement, one generally assumes that the inputs are fixed. One can formulate an analogous property for other types of protocols, particularly those whose decisions are in some sense irreversible. For example, if the goal of a protocol is to agree when to commit updates or garbage collect them, then a decision, once acted upon, can no longer be undone.

be logical continuity, and some strong form of consistency in the entire flow of information that is passed "upwards" from P_B, whether on B_3 or B_4. In our approach, we refer to such a flow as a *Distributed Data Flow* (DDF) [6, 8]. Conceptually, Q operates on DDFs passed up from P_A, P_B, and P_C instead of dealing with the individual nodes. The failure of B_3 and the arrival of B_4, B_5 can thus be dealt with by program P_B, e.g., by state transfer or similar techniques while Q continues to consume/transform what logically remains the same flow.

In our technical report [7], we describe a protocol language that supports the flavor of recursion discussed above, and in which DDFs play role analogous to function arguments and local variables. Programs in this language can be translated into hierarchical architectures described in our past work [8]. Due to limited space, here we just briefly discuss an example recursive algorithm expressible in it (more details are in [7]).

Figure 1 shows a program for hierarchical leader election. The input *candidate* is a distributed flow of proposals whom to elect, each represented as an integer identifier. A different identifier might be submitted into the protocol at each node at a different time, but we treat them all as parts of a single logical "upwards" input flow of proposals. Likewise, there is a single "downwards" output flow *leader* carrying the results of election. Proposals and results are not one-offs; they flow continuously in and out, for as long as the protocol is active.

Internally, directive **independently** allows the flow of proposals to be partitioned into N subflows, and recursively instantiate our hierarchical election protocol on each of these. Recursion terminates with the **where** (**singleton**) clause, on subflows confined to only a single location. The outcome of the recursive invocations is labeled as *local_leader* and fed as a parameter into yet another protocol **stable_elect**, which combines partial results in *local_leader* into the output flow. Protocol **stable_elect** is not recursive; we list it for completeness, but do not discuss it here due to limited space (see [7]).

Since **stable_elect** was not prepended with **independently**, it runs on all nodes, on the entire internal flow *local_leader*. It might seem as though we have not gained much. However, we know that the subflow of *local_leader* within each of the N partitions is consistent (this is the meaning of type qualifier **s-up** in line 01; for details, see [7]), so only one node in every partition needs to pass the information from *local_leader* into **stable_elect**. This is illustrated on Figure 2. Here, software components $A_i^{(1)}$ shown as large, white boxes run **elect** on a full set of 4 nodes. Internally, subcomponents $A_j^{(2)}$ (bottom left, yellow) run an embedded instance of **elect** on 2 of these nodes, and $A_k^{(3)}$ run **elect** on the other two. Finally, all $B_l^{(1)}$ run **stable_elect**. Arrows represent the flows of information. The gray arrows represent redundant outputs of **elect**; there is no need for 2 of the 4 nodes to participate in **stable_elect**. A similar analysis could be extended to multiple levels in a hierarchy, to show that each instance of **stable_elect** in this program only needs to run at $O(1)$ nodes. The analysis and deployment could be performed automatically by a compiler.

3. CONCLUSIONS

We have shown how trying to represent a hierarchical algorithm such as agreement using recursion leads to the concept of a DDF as a first-class language abstraction. We discussed an example algorithm in a DDF language, and we introduced a new flavor of static analysis that can use DDF's distributed types to generate a more scalable runtime architecture.

```
01: object elect(up int candidate) : s-up int leader {
02:   where (singleton) leader := candidate;
03:   elsewhere {
04:     up int local_leader := independently elect(candidate);
05:     leader := stable_elect(local_leader);
06: }}
07: object stable_elect(up int candidate) : s-up int leader {
08:   s-up int elected := 0;
09:   where (fresh elected ∧ elected ≤ candidate)
10:     elected := min candidate;
11:   leader := elected;
12: }}
```

Figure 1: Leader election expressed using recursion.

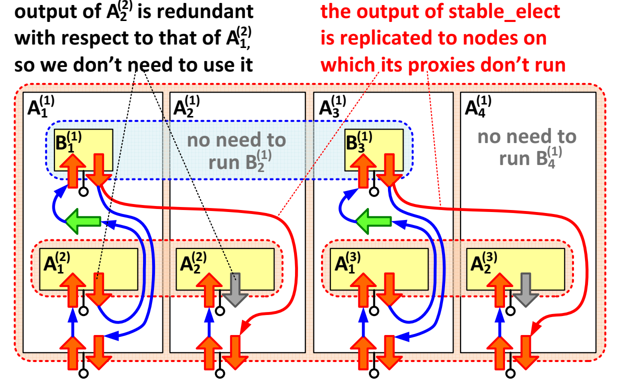

Figure 2: Protocol stable_elect need not run on every node because the output of elect is consistent within each partition in which it is recursively instantiated.

4. REFERENCES

[1] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. *Technical Report CNDS-98-4, Johns Hopkins University*, 1998.

[2] K. Guo, W. Vogels, and R. van Renesse. Structured Virtual Synchrony: Exploring the Bounds of Virtual Synchronous Group Communication. *7th ACM SIGOPS European Workshop*, 1996.

[3] Q. Huang, D. Freedman, Y. Vigfusson, K. Birman, and B. Peng. Kevlar: A Flexible Infrastructure for Wide-Area Collaborative Applications. *11th International Middleware Conference (Middleware 2010)*, 2010.

[4] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A Group Membership Service for WANs. *ACM Transactions on Computer Systems*, 20(3):191–238, February 2002.

[5] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. *20th ACM Symposium on Operating Systems Principles (SOSP 2005)*.

[6] K. Ostrowski, K. Birman, and D. Dolev. Distributed Data Flow Language for Multi-Party Protocols. *5th ACM Workshop on Programming Languages and Operating Systems (PLOS 2009)*.

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

[8] K. Ostrowski, K. Birman, D. Dolev, and C. Sakoda. Implementing Reliable Event Streams in Large Ssystems via Distributed Data Flows and Recursive Delegation. *3rd ACM International Conference on Distributed Event-Based Systems (DEBS 2009)*.

[9] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. Macedon: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI 2004)*.