MAD Ensembles Are Saner Than You Think

Petros Maniatis, Intel Research Berkeley, USA

1. INTRODUCTION

Today's and tomorrow's multiprocessing or distributed computing deployments comprise large ensembles of elements such as processors in data centers, sensors in a monitoring deployment, or even actuators in a robotic swarm. Increasingly, these ensembles are federated across administrative domains: one can readily patch his very own mini PlanetLab to *the* PlanetLab and exchange resources at a looser contractual connection than via full consortium participation; more pervasively, the Internet operates thanks to the ability of over 30000 (competing) organizations to interconnect their router ensembles in a patchwork of autonomous forwarding systems. In the near future, it is not implausible to imagine two competing companies that federate their data analysis resources to grow their aggregate network monitoring reach, or a reluctant military alliance that combines unmanned aerial vehicle fleets over a common staging area to increase observational accuracy.

Nevertheless, programming such ensembles typically focuses on the individual, specifying the behavior of each as if it were an isolated, lone computer. Though engineers have survived this practice at small scales, they are likely to encounter towering hurdles when programming larger, more heterogeneous, and less mutually trusting ensembles, for several reasons.

First, an ensemble's goals tend to be *global*, whether they involve the task to be performed ("track the path of each packet through Internet routers") or any notion of quality and correctness to be achieved ("obtain at least 90% of a path with probability at least 90%"). Translating manually global specifications to local actions is not only complex and extremely error-prone, but also rife with subtle opportunities for rational or Byzantine misbehavior, resulting in brittle systems and unenforceable service level agreements.

Second, however skilled a programmer is at mapping global goals to individual elements' behaviors under given expected conditions, in practice conditions change, often requiring a different approach from the one assumed by the programmer. For example, a loosely coupled monitoring infrastructure that is well built for a higher jitter network may be entirely ill-suited to a network that trades off higher loss rates for lower jitter. Given the business-dictated need for opacity across administrative boundaries, a monitoring participant would be left with two equally unsatisfactory choices: either use a common, globally sanctioned as "correct" implementation that handles a middle-of-the-road network suffering the inefficiencies involved, or build a custom, efficient implementation with the requisite cost of convincing its peers it is not cheating.

Third, even if all convincing can be done, customized implementations of a shared task or protocol may fail, violating the agreed-upon correctness conditions in different and interesting ways, making it almost impossible to reason about how to recover compliance – e.g., incrementally find an alternate computation path to complete the task – without restarting from scratch.

Layering has been the textbook solution to such problems before: specify common interfaces for each functionality class and leave the implementation details, including local optimizations, to the implementers. Unfortunately, the difficulty of producing good, standards-compliant implementations of layers usually leads to one-size-fits-all solutions that either do more than is necessary (e.g., mandate communication over TLS even between directly connected, mutually trusted parties) or less than is sufficient (e.g., forgo authentication among BGP routers altogether because not everyone belongs to a common Public Key Infrastructure). What is more, layering usually imposes a rigid stack of functionalities (application on top of transport on top of routing) which often conflicts with what the system designers could optimize for (e.g., application-controlled routing on top of transport) [3].

2. ENSEMBLE PROGRAMMING FOR MAD SYSTEMS

The thesis of this note is that the problem lies in the venerable habit of designing ensembles from the bottom up, from the layer, to the stack, to the computer, to the population. We advocate instead that to design, build, and monitor large, multiple-administrative-domain computing ensembles we need top down programming frameworks. In such a framework, the global task and its desirable properties are explicitly and systematically specified by the programmer in some formalism. One approach is to treat the whole ensemble as a single, gigantic computer made up of the ensemble's individual components. The specification can be checked automatically against its global conditions of quality or correctness within this abstract environment (as per traditional theorem proving systems); if the check fails at this highest level of abstraction, there is probably no need to start discussing transport protocols, specific cryptographic primitives, or processor power levels.

Translation from the global specification to more concrete, lower-level specifications can be done automatically, much like compilation for typical programs or planning for database queries, perhaps at multiple intermediate granularities. For example, it may be helpful to partition the global abstract computer to a set of administrative-domain-specific components, since under most scenarios, what happens within an administrative domain is not only customized to fit local needs, but may also be private. The breakdown of the high-level functionality and required properties into domain-local functionality and properties also allows a relaxation of the threat model from a full-blown MAD environment across domains to an honest-operator-faulty-software model within a single domain. Knowledge of particular domain interrelationships such as degrees of trust can inform this refinement on a case-by-case basis. Similarly, the translation can move to smaller subgroups, such as network locality groups to optimize communication patterns, single-rack groups to optimize heat dissipation, and single processor cores to take advantage of shared processing structures. At the bottommost level, the ensemble's individual elements are programmed, each in its own machine language.

Along the translator, an optimizer can be invaluable in this process, both during "compile time" and during runtime tuning. Whereas translation occurs *vertically* through refinement, optimization can operate *horizontally*, collapsing redundancy, rearranging computations to improve communication patterns, changing implementations of packaged functionalities (e.g., from a proactive to a reactive routing protocol) to match impedance with other choices (e.g., reactive routing with lazy consistency enforcement).

Finally, the execution engine can take the final runtime, verify any proofs of correctness supplied by the translator and optimizer, and execute the system, ensuring that adaptation is triggered as conditions change.

The need for explicit, portable assurances, especially within the context of MAD ensembles, is unavoidable and challenging through all stages of the process. At the specification stage, the language must be expressive enough to describe access and flow control restrictions (e.g., "tax IDs should not flow through ISP X's wires in the clear"), and priorities among different optimization axes (e.g., "use as little memory as possible and then protect from information leak to the extent achievable"). At the optimization stage, it must be possible to keep track of how each program transformation – partitioning, replication, or reordering – affects the constraints of the specification (e.g., split a private, two-party computation without leaking information across the parties). At the execution stage, the system must enforce the semantics of the specification, e.g., ensuring that private information is not placed in the clear on shared memory; it must also make it possible to debug, so that a programmer can figure out what is going on without having to compromise any guarantees made by the system (e.g., restart a deadlocked, privacy-preserving computation without having to compromise the privacy of inputs or partial outputs).

Deep down, this vision is founded on a simple (repeatedly rediscovered) principle: specifications should be layered but runtimes need not be. Structure among functionalities helps human designers separate their concerns in relevant units; however, the realities of performance and fault-tolerance optimizations break layer boundaries invalidating the benefits of abstraction. Perhaps it is time to reconsider having the cake (of layered specifications) and eating it too (by producing safe yet optimized, layerless runtimes).

3. MAD DECLARATIVE ENSEMBLES

We have been pursuing this agenda in the context of the P2 Declarative Networking Project [1]. In P2, a distributed application is expressed declaratively in OverLog [5], a logic language based on Datalog. The whole ensemble is treated as a distributed database, its algorithm expressed as a continuous query over all elements' changing states. A compiler translates the logic into event-condition-action rules performed on single elements, optimizes them to minimize data movement (a la System R's optimizations [8] with network costs included), and plans the resulting runtime as a software dataflow graph, similar to Click [4]. Execution interprets the dataflow graph.

Our current research in the P2 project pursues the goals of the MAD ensemble programming agenda, along multiple paths. First, on the language front, we are growing P2's declarative specification language (OverLog) to include primitives about authentication (e.g., Binder's "says" construct [2]), information flow (such as decentralized labels [6]), and consistency requirements (such as linearizability or session guarantees). This allows OverLog's expressivity to capture not only algorithmic specifications but also the security and fault-tolerance requirements of the specified application.. Furthermore, we are strengthening OverLog's type system to make the language type safe.

Second, we are evolving the P2 compiler and optimizer so that, in addition to dealing with the new language features above, they can provide assurances on the correctness of output runtimes. We are exploring proof-carrying code [7] as an approach to allow programs from potentially untrusted compilers (especially during runtime adaptation) to be shared across domains. P2's logic-based specification language enables a straightforward connection to a theorem prover for the generation of proofs that go beyond simple memory safety to global properties such as resource quota compliance. We are also adapting the techniques of secure program partitioning [9] to our compiler's existing (insecure) program partitioning transformations, taking advantage of any information flow annotations. The compiler also encapsulates parts of a program's specification in known fault-tolerant constructs such as Byzantine-fault tolerant replicated state machines to ensure strict consistency guarantees for sensitive parts of a specification.

Third, we are migrating the remaining imperative components of the runtime (as well as the debugging and adaptation modules) to a type-safe system language (we are currently considering OCaml); large portions of the runtime adaptation module as well as the compiler and optimizer are themselves written in OverLog, simplifying this process. The goal is to eventually make the whole P2 software stack certifiably memory-safe.

Though exciting and promising, these early steps do not make the scope of the MAD ensembles agenda any less ambitious (or mad). Our path is not dictated by feasibility but, rather, by necessity. As cookbook solutions to specific problems for untrusted environments mature, thanks in the largest part to the efforts of the distributed systems and dependability communities, we hope our findings will help to give programmers the tools they need to design, program, verify, and debug large, complex, heterogeneous deployments across administrative boundaries.

4. REFERENCES

[1] The P2 Project.
`http://p2.berkeley.intel-research.net/`.

[2] M. Abadi and B. T. Loo. Towards a Declarative Language and System for Secure Networking. In *NetDB*.

[3] T. Condie, J. M. Hellerstein, P. Maniatis, S. Rhea, and T. Roscoe. Finally, a use for componentized transport protocols. In *HotNets*, 2005.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM TOCS*, 18(3), 2000.

[5] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[6] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Software Engineering and Methodology*, 9(4), 2000.

[7] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *OSDI*, 1996.

[8] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[9] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, 2001.