A Discreet, Fault-Tolerant, and Scalable Software Architectural Style for Internet-Sized Networks

Yuriy Brun
Computer Science Department
University of Southern California
Los Angeles, California 90089, USA
ybrun@usc.edu

Abstract

Large networks, such as the Internet, pose an ideal medium for solving computationally intensive problems, such as NP-complete problems, yet no well-scaling architecture for Internet-sized systems exists. I propose a software architectural style for large networks, based on a formal mathematical study of crystal growth that will exhibit properties of (1) discreetness (nodes on the network cannot learn the algorithm or input of the computation), (2) fault-tolerance (malicious, faulty, and unstable nodes cannot break the computation), and (3) scalability (communication among the nodes does not increase with network or problem size). I plan to evaluate the style both theoretically and empirically for these three properties.

1. Research Problem and Motivation

The Internet's growth has created networks with great computing potential without a clear way to harness that potential to solve memory- and processor time-intensive problems. Networks, such as the Internet, have the potential to solve NP-complete problems quickly, but as their individual nodes may be unreliable or malicious, users may desire guarantees that their computations are correct and are kept confidential.

My work is particularly applicable to problems that are computationally intensive and easily parallelizable. Computationally intensive problems are ones that a single computer is unlikely to solve quickly, while easily parallelizable problems are ones that inherently yield a large number of parallel threads. For example, all NP-complete problems have both of those properties [3]. Further, my work is applicable to users that desire discreetness and have access to large but unreliable networks. By discreetness, I mean that the user does not want others to find out the input or the algorithm. By large but unreliable network, I mean a network, such as the Internet, that is partially or entirely outside of the user's control, and perhaps even hostile. I describe two scenarios that are the heart of the problems I am tackling.

Scenario #1: A large university wishes to digitally deliver recent graduates' transcripts to graduate schools and employers. This information is sensitive, and needs to be encrypted using the recipient's public key and digitally signed using the university's private key. It may take the university's transcript department's computer months to encrypt and sign the thousands of requests, but fortunately it has access to the university's network of computers. While that network may be large, the individual nodes are insecure and cannot be trusted with sensitive data such as the university's private key or the students' transcripts.

Scenario #2: An espionage agency is attempting to break an RSA code sent by an enemy. The agency wishes to factor the enemy's public key; however, it cannot allow anyone to know the key's factors or even whose key it is factoring. Since the agency has access to the Internet, it is feasible to factor nondeterministically, or through brute force. However, the problem is to do so discreetly, without the nodes on the network learning the problem or the input.

Both these scenarios will result in complex distributed software systems. It has been shown that such systems are most effectively approached from an architectural perspective (e.g., [2]). In particular, software architectural *styles* present generic design solutions that can be applied to problems with shared characteristics.

I propose a software architectural style that allows distributing problems over a large network in a fault-tolerant, discreet, and scalable manner. To that end, I will rely on a formal model of crystal growth [4]. This model is Turing universal, thus it can compute all the functions that a traditional computer program can. Systems in this model show remarkable fault-tolerance, distribution of information, and scalability, and a software architecture that implements the rules of such systems should inherit these properties. I plan

29th International Conference on Software Engineering (ICSE'07 Companion)
0-7695-2892-9/07 $20.00

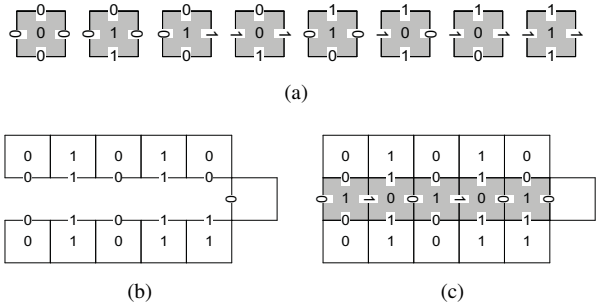

Figure 1. A sample tile system that adds numbers. (a) The system has eight computation tiles. (b) A seed configuration encodes the inputs, $10 = 1010_2$ and $11 = 1011_2$. (c) The gray computation tiles attach to the seed to form the output $21 = 10101_2$.

to evaluate the architectural style theoretically, mathematically analyzing the architecture, and empirically on a system that solves NP-complete problems.

2. Computing with Tiles

It is somewhat counterintuitive that tile systems can compute complex functions because tiles are limited to local interactions. However, while the individual tiles are simple, together they become as powerful as every computer. Figure 1 shows an example of a tile system computing the sum of $10 = 1010_2$ and $11 = 1011_2$.

The adding system has eight computational tile types (Figure 1(a)). The center state variable of each tile in Figure 1(c) represents one bit of the solution, and the west side represents the next carry bit. Starting from a seed configuration (Figure 1(b)), instances of the gray tile types can attach if their sides match the neighbors' sides. The final configuration (Figure 1(c)), encodes the answer $21 = 10101_2$ in the center row. I have designed complex systems that multiply [1], factor, and solve NP-complete problems.

A tile style architecture is based on a tile system. The components of the architecture are instantiations of the tile types. While a system based on this architecture will have a large number of components, there are only a small number of different *types* of components (e.g., eight types for adding). Nodes on the network represent these components, and components that are adjacent in an assembly can *recruit* other components to attach. Note that many components (i.e. tiles) can run on a single physical node.

Each component's external *structure* is a state variable (shown in the center of each tile) and four *interfaces*, i.e. side variables (shown on the sides of each tile). The location in the assembly may also be useful for recruiting. The *topology* is a 2-D grid of components that allows neighbors on the grid to interact. The components exhibit two *behaviors*: cooperating with neighbors to recruit suitable new components to attach, and reporting the solution to the user. Recruitment is the principal functionality performed by a given tile. The *interaction* consists of exchanging data about a component's sides in order to recruit, and the *data flow* is limited to the components' state variable and sides.

A user who wishes to solve a computationally intensive and easily parallelizable problem, e.g. an NP problem, and has access to a large network, may use the tile architectural style to design a system to solve her problem. The user has two options: design her own architecture based on the tile style to solve her particular problem, as I describe in [1], or reduce her problem to SubsetSum, using a standard polynomial time reduction [3], and use the SubsetSum architecture I am developing as part of my dissertation. Whichever tile system the user chooses will serve as the template for the architecture: the system's tiles defining the types of components. Part of a tile system is the description of seeds that encode inputs (e.g., Figure 1(b) shows the seed for adding 10 and 11). The user sets up a seed to encode her input and assigns computers on the network to represent the seed tiles. Once the initialization is complete, starting with the seed tiles, adjacent components recruit other nodes to represent fitting components and eventually produce the solution.

In the addition example, each component represents a single bit of the solution. It is possible to have individual components represent larger chunks of data, limiting the necessary network communication, thus yielding a trade-off between discreetness, fault-tolerance, and communication.

3. Contributions

The main contribution of my research is a scalable and fault-tolerant software architectural style for discreet computation on a large network. To that end, I have extended a theoretical model of self-assembly by defining the notion of computing functions, designed tile systems that compute functions deterministically (e.g., adding and multiplying) and nondeterministically (e.g., factoring), and proven their correctness and bounds on fault-tolerance, self-regeneration, and probability of success of computation. I have also created a preliminary design of the software architectural style based on the tile assembly model.

References

[1] Y. Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Computer Science*, 10.1016/j.tcs.2006.10.025, 2006.

[2] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[3] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[4] E. Winfree. Simulations of computing by self-assembly of DNA. Technical Report CS-TR:1998:22, Caltech, 1998.