Preserving Privacy in Distributed Computation via Self-Assembly

Yuriy Brun and Nenad Medvidovic, *Member, IEEE Computer Society*

Abstract—We present the tile style, an architectural style for distributing computation onto large, insecure, public networks, such as the Internet. The tile style provides guarantees on (1) privacy preservation: tile-style systems preserve the privacy of the algorithm and data, (2) fault and attack tolerance: tile-style systems can tolerate faulty and malicious nodes, and (3) scalability: tile-style systems scale well to leverage the size of the public network to accelerate the computation. We concentrate on systems that solve NP-complete problems and demonstrate how tile-style systems can solve important real-world problems, such as protein folding, image recognition, and resource allocation. We present the algorithms involved in the tile style and formally prove that tile-style systems preserve privacy. We develop Mahjong, a tile-style implementation, and empirically evaluate it on several physical networks of varying sizes, including the globally distributed PlanetLab. Our analysis demonstrates tile style's scalability and ability to handle varying network delay, and shows that problems requiring privacy-preservation can be solved using the tile style orders of magnitude faster than using today's state-of-the-art alternatives.

1 Introduction

Today's best known methods for solving certain important computational problems require exponential time. Single computers can practically solve only small instances of such problems. Large networks, such as the Internet, have the potential to solve larger instances significantly faster. The ability to solve moderate-sized instances of such problems hundreds of times faster than a single machine has substantial academic, financial, and social implications and greatly impacts such fields as medicine, management, and systems engineering. For example, the ability to determine proteins' minimal-free-energy structures within days (as opposed to years) could lead to cures or treatments of cancers, HIV, and other life-threatening diseases. The ability to predict the optimal allocation of resources could cut costs of public and private projects dramatically and assist in properly planning for such large endeavors as Boston's Big Dig.

Existing approaches that combine Internet's computational resources to solve complex problems leverage one of the following: (1) a master-worker model (e.g., SETI@home [38] and Folding@home [42]); (2) a centralized scheduler with intricate knowledge of the entire system's state [9], [33]; or (3) decentralized scheduling and distribution mechanisms [21], [25]. Some of these approaches have resulted in commercial enterprises and have been deployed on over a million machines [53], providing results to some of the largest computational problems solved to date in the areas of artificial intelligence [19], physics [41], and neuroscience [18]. The organization of loosely coupled networked devices has permitted advances in science and engineering that would otherwise not be possible [10], [36], [43].

However, the centralized nature of master-worker and centralized-scheduler systems make them unable to scale to Internet-sized networks. The decentralized scheduling and distribution approaches, such as MapReduce [25] and the organic grid [21], provide steps in the proper direction toward a highly scalable computation-distribution platform. While these approaches appropriately focus on decentralization and scalability, they rely on secure and fairly reliable underlying computational resources. Unfortunately, Internet-sized networks of public, volunteer computers are vastly unreliable and insecure.

In this paper, we propose the *tile architectural style*, a decentralized approach to distributing computation onto large public networks. The tile style leverages mechanisms from the nature's process of self-assembly and inherits a number of nature's properties, such as decentralization, tolerance of faults and attacks, and localization of information. By using nature's mechanisms, tile-style systems can leverage extremely large public networks and handle failures and attacks on its nodes and the high churn rate typical in such networks.

While the tile style can compete with existing techniques in a number of dimensions, in this paper, we focus on comparing scalability and efficiency, while demonstrating the tile style's unique ability to guarantee the privacy of the computation's data. We have previously discussed the tile style's ability to handle faults and malicious attacks [17], [16], and do not focus on that dimension here. While the tile style can be used to distribute a wide range on computations, we focus our discussion here on NP-complete problems, a large and important class of problems.

Our approach benefits in two ways from being an

- *Y. Brun is with the Computer Science and Engineering Department, University of Washington, Seattle, WA, 98195. E-mail: brun@cs.washington.edu*
- *N. Medvidovic is with the Computer Science Department, University of Southern California, Los Angeles, California, 90089. E-mail: neno@usc.edu*

architectural style. First, the tile style abstracts away the details of the underlying tile model. This abstraction allows an architect who is unfamiliar with the underlying tile mechanisms, but understands software architectural concepts, such as components and connectors, to develop tile-style systems to solve arbitrary NP-complete problems. The underlying tile mechanisms are "under-the-hood" details used to prove properties of the tile style. Second, as architectural styles present generic design solutions that can be applied to problems with shared characteristics [50], the tile style allows for the development of systems that solve distinct computational problems, while uniformly handling nonfunctional concerns, such as reliability, privacy, and scalability.

The tile style decomposes an algorithm into some of its most basic operations and assigns individual nodes on a network to deploy objects representing each of the input and intermediary data bits. The objects then communicate over the network to compute the results, in some sense self-assembling the computation. These simplistic objects are analogous to molecules, floating around in solution, attaching and self-assembling to form crystals. This extensive decomposition allows for the objects to be highly localized, requiring interactions only with immediate neighbors. This self-assembly process requires significant network communication. However, because the tile style targets NP-complete problems, which have a large (exponential in the size of the input) number of independent threads, many nonblocking computations can be executed in parallel. This ensures that no node ever waits for network communication, allowing tile-style systems to leverage network size to enhance the speed of the computation.

In order to demonstrate the computational feasibility of our solution, we provide Mahjong, a reference implementation of the tile style in the form of a middleware platform. We deploy Mahjong on three distinct networks, including the globally distributed testbed PlanetLab [48]. We also simulate Mahjong's execution on virtual networks of up to 1,000,000 nodes. We empirically verify that the speed of tile-style computation is proportional to the number of nodes and that network delay has little-to-no effect on the speed. Furthermore, we formally analyze the communication and computation costs induced by the tile style, provide bounds on their time requirements, and then use Mahjong to verify them empirically. Finally, we formally prove that tile-style systems preserve the privacy of the data used in the computation as long as no adversary controls more than one half of the public network — a highly unlikely occurrence.

Our analysis shows that, for networks larger than about 4000 nodes, the tile-style solutions outperform today's state-the-art privacy-preserving solutions. For example, problems that today take 4 years to solve can be solved using a tile-style solution deployed on a network the size of SETI@home (1.8 million nodes [53]) in 7 days.

The rest of this paper is structured as follows: Section 2 presents several problems tile-style systems target. Section 3 positions our work in terms of related research. Section 4 explains how computation is possible in the tile assembly model. Section 5 describes the tile architectural style. Section 6 discusses our implementation and a set of experiments designed to demonstrate tile style's feasibility. Section 7 formally analyzes tile style's privacy-preservation. Finally, Section 8 summarizes the paper.

2 Research Scope and Target Problems

To identify our research contributions, in this section, we summarize the scope of our work and present two representative problems our technique helps to solve.

2.1 Research Scope

Our goal is to provide an approach for aggregating distributed computational resources into a highly robust system. The resulting system benefits from the Internet's large size while outperforming existing techniques by tolerating the insecurity and unreliability inherent to the Internet. The primary contributions of this paper are:

- The tile style: a software architectural style for distributing NP-complete problems onto large distributed systems while (a) preserving the privacy of the computation, (b) scaling well to leverage the resources, and (c) tolerating faulty and malicious nodes.
- A formal theoretical analysis of the tile style's privacy preservation, efficiency, and scalability.
- Mahjong: an implementation of a tile-style system.
- An empirical evaluation of Mahjong, deployed on a wide range of distributed testbeds, including the globally distributed PlanetLab.

While we make sizable steps toward our goal, it is important to note that we designate several directions of research as outside of the scope of this paper. First, we do not consider issues that are common to all Internet-based systems. For example, the energy consumption of an Internet-sized computing network is important, but is not unique to our approach and has been argued non-prohibitive in volunteer computing literature [2]. Second, we do not focus on programming models and language-level solutions that can be envisioned as natural outgrowths of this work. In particular, it is relevant to discuss a tile-based language that would expand the tile style's application outside of NP-complete problems, but we leave this direction as part of our future work. Third, while we focus on scalability, efficiency, and privacy, there are a number of dimensions of Internet-sized computing systems outside of this paper's scope, including energy efficiency (mentioned above) and fault-tolerance (which we discussed in [17], [16]). Where applicable, we will provide intuition for the tile style's ability to compete with existing techniques in these dimensions.

2.2 Target Problems

Existing techniques that use the Internet to solve complex computationally-intensive problems [25], [38], [42] do not take into account that data involved in these computations may be sensitive and that network nodes may be malicious and may attempt to extract private data from the computation. In fact, these techniques make it trivial to learn the nature of the computation and the relevant data. The tile style, our proposed technique for distributing NP-complete computation, is *privacy-preserving*: the involved data remain private during and after the computation. The following are two example scenarios the tile style targets.

1. A pharmaceutical company has generated a series of candidate proteins for treating a particular cancer. The company needs to predict the 3-D structure of the proteins as they would fold within the human body but the proteins' amino acid sequences are valuable intellectual property and must remain private. The protein-folding problem is NP-complete [8], and thus for reasonably-sized proteins, it could take years on a single computer, or even on small private networks, to compute the desired structures. The company is unwilling to use existing approaches [42] to distribute the computation on a public network because these approaches distribute the amino acid sequences to all helping nodes.

2. Image recognition is at the heart of many advanced artificial intelligence and security tasks. Matching faces captured by a camera to a database of known criminals allows automated intruder detection and aids security at public locations such as airports and casinos. However, facial recognition and image matching problems are NP-complete [37] and many people may enter the location of interest at once. Further, any employed solution must execute quickly to deliver results in real-time. In order to protect the identity and privacy of the innocent individuals entering the location, the system must either guarantee that the entire computation takes place on a completely secure private network, or use a privacy-preserving technique to distribute the computation on a public network.

3 Related Work

In this section, we describe related work in the areas of software architecture, distributing computation onto untrusted hosts, and privacy-preserving computation.

3.1 Software Architecture

Software architecture has been identified as an important part of building almost all large systems. Architecture is the set of principal design decisions about a software system [52]. Architectural styles can be used to "force" a software system to conform to certain rules, thus resulting in some desired properties. By constraining the structure, behavior, interaction, and composition of components, styles help ensure the system's key properties, such as performance, reliability, portability, scalability, and interoperability [52]. For example, mandating that two components communicate via implicit invocation can result in systems that are more easily evolvable. However, it is also possible to provide desired system properties as an emergent behavior of the architecture without forcing restrictions on the system designer. Several researchers have argued that for a system to be self-healing, the system must be self-observant and alter its behavior in hostile environments (e.g., as overviewed by Kramer and Magee in [39]). However, in our proposed tile architectural style, the system exhibits properties of self-healing naturally, without observing or altering its behavior. Similarly, Devanbu and Stubblebine have argued that security, a crucial property of most modern software systems, may be implemented in the connectors mediating the interactions among the system's components [26]. Accordingly, the tile style, in principle, allows for security in the connectors; however, privacy preservation, one aspect of security, is an *emergent* property of the style.

There has been at least one other attempt at providing architectural models inspired by natural phenomena and processes: Inverardi and Wolf leveraged the chemical abstract machine (CHAM) to build natively evolvable models of software systems inspired by chemical reactions [35]. The CHAM model's objective was to demonstrate that real systems' architectures can be captured and analyzed for certain properties, and that the architectural models expressed in CHAM can easily evolve. Even though it was one of the early proposed software architectural models, CHAM has proven to be solely of academic interest, most likely because it was never accompanied by implementation support. In contrast, the objective of the tile style is to produce models that solve concrete problems with real-world applicability, in a manner that deals with several critical properties of those problems, and to support mapping those models to effective distributed implementations.

Finally, while there are several definitions of architectural styles (e.g., [52], [6], [23], [50]), we will directly leverage Mikic-Rakic et al.'s [45] definition in formulating the tile style. This definition is useful in that it provides an operational view of a style's building blocks. Mikic-Rakic et al. have argued that an architectural style can be described along five dimensions: external structure, topology rules, behavior, interaction, and data flow. External structure describes the "outside view" of the components adhering to the architectural style; topology rules describe the allowed paths of interaction among those components; behavior describes the components' internal function and state; interaction captures the collaboration between the components; and data flow specifies the structure of the data exchanged by the components. We will follow this scheme in defining the tile architectural style in Section 5.

3.2 Getting Help with Computation

The growth of the Internet has made it possible to use public computers to distribute computation to willing hosts. Software designed to solve computationally intensive problems has emerged to take advantage of this phenomenon, enticing users to devote their computers' idle cycles to some academically or otherwise worthy cause. This notion focuses the underpinning of computational grids [29]. Among systems that concentrate on distributed computation are BOINC systems [2] (such as SETI@home [38] and Folding@Home [42]), MapReduce [25], and the organic grid [21]. A unique approach — FoldIt — uses the competitive human nature to solve the protein-folding problem [3]. FoldIt asks humans, as part of a game, to try to arrange a protein's amino acids to minimize the free energy. The hope is that humans will be more efficient than the brute force approaches and that a large number of users will help find the optimal solution. These systems try to solve exactly the highly parallelizable problems toward which our work is geared, but unlike the tile style, they do not preserve privacy.

Cloud computing is a relatively new phenomenon that allows outsourcing computation. Corporations such as Google, Yahoo!, and Amazon have the computational resources to distribute these computations onto thousands of privately-owned, networked, fairly reliable machines. Clouds leverage technologies such as MapReduce [25] to handle the data and computation distribution. Today, a MapReduce system running on a 10,000-core cluster produces data used in every Yahoo! web search query [4]; as many as a thousand MapReduce jobs are executed on Google's and Amazon's clusters daily [1], [25]; and Facebook uses a MapReduce system to process more than 15 terabytes of new data every day [57]. While commercially viable, clouds rely on legal contracts to ensure privacy. For example, if a pharmaceutical company outsources a protein-folding problem to Google, that company must share the valuable amino acid sequence with Google and be protected from Google misusing the sequence or making that data public by a contract. Our approach allows the pharmaceutical company to distribute its problem on several clouds without having to disclose the private data, while providing guarantees that the operators of these clouds, as well as potential attackers, cannot compromise that data.

Some research, rather than leveraging large networks, has attempted to accelerate NP-complete computation by developing faster algorithms for single machines and small clusters. This work ranges from developing efficient exponential-time algorithms [40], [55], to using runtime information to dynamically improve the speed of SAT solvers [5], to leveraging local message-passing protocols such as MPI and OpenMP to use small clusters to linearly accelerate the computation of specialized problems [47]. This work is not in competition with our technique, but is rather complementary. The tile style is based on a Turing-universal computational model [54] and can implement each of these advanced algorithms on large distributed networks, leveraging both their efficiency and the tile style's privacy preservation, scalability, and fault tolerance. At times, in this paper, we will compare simple algorithms that solve NP-complete problems implemented using the tile style versus using conventional methods. The same comparisons can be made for complex, efficient, state-of-the-art algorithms.

The majority of research on strategies for getting computational help without disclosing the input of the computation has focused on asking a single other computer for help. Yet, in classical (as opposed to quantum) computing, it is not possible to get help from a single entity in solving an NP-complete problem without disclosing most of the information about the input and the problem one is trying to solve [22]. Our approach consists of distributing such a request over many machines without disclosing the entire problem to any small-enough subset of them.

3.3 Secure Computation

Gentry has theorized about using a fully homomorphic encryption scheme to encrypt a circuit describing a problem and then executing the encrypted circuit on a separate agent without disclosing the private data [30]. While theoretically exciting, practically, this approach cannot be used today because of the exponential amount of computation required to encrypt and decrypt. In a popular article, Gentry himself estimates that using his technique to perform a Google search, while keeping the query private, would require one trillion times as much computation as is needed today [32]. Gentry's technique is theoretically more powerful than ours because it keeps the data private from the entire network (as opposed to subsets of the network). However, as we demonstrate in Section 6, unlike homomorphic encryption, our technique is efficient enough to be used today.

The field of secure multi-party computation explores whether multiple computers, each of whom knows part of an input, can compute a function of that entire input without sharing their parts with others. The tile style is a solution to a related, but fundamentally different problem: can computers be used to help compute a function if no sizable group of those computers knows the input to the computation? The seminal work in the area of secure multi-party computation introduced Yao's garbled circuit protocol that allows n nodes, each with access to a single input, to compute a function of the n inputs while disclosing only the value of the function to each node [56]. Zero-knowledge compilers bridge that work closer to our approach by making Yao's protocol secure even if the parties cannot be trusted [31]. Secure multi-party computation applies to functions on large distributed private data sets, while our work applies to functions on fairly small data sets, but ones that require exponential time or space to compute. Our work does, at

times, leverage some of the work in secure multi-party computation, as we describe in Section 5.3.

4 Computing with Tiles

The tile assembly model is a formal model of molecular self-assembly that describes how simple molecules can form complex crystals. In this model, molecules are square tiles with special labels on their four sides. Tiles can stick together under certain conditions when their abutting sides' labels match. Winfree showed that this process allows molecules to compute functions and proved that the model is Turing universal [54]. However, Winfree acknowledges that using his approach to solve computational problems produces highly inefficient assemblies and resembles programming using Turing machines. We have continued Winfree's work by developing the notion of efficient computation within the tile assembly model and constructing efficient assemblies to add and multiply integers [12], factor integers [13], and solve NP-complete problems [14], [15]. Section 4.2 will describe the assembly that solves 3-SAT.

4.1 Theoretical Underpinnings

The tile assembly model has *tiles*, or squares, that stick or do not stick together based on various *interfaces* on their four sides. Each tile has an interface on its top, right, bottom, and left side, and each distinct interface has an integer *strength* associated with it. The four interfaces, elements of a finite alphabet, define the type of the tile. The placement of a set of tiles on a 2-D grid is called a *crystal*; a tile may *attach* in empty positions on the crystal if the total strength of all the interfaces on that tile that match its neighbors exceeds the current *temperature*. Starting from a *seed crystal*, tiles may attach to form new crystals. Sometimes, several tiles may satisfy the conditions necessary to attach at a position, in which case the attachment is nondeterministic. A tile assembly $\mathbb{S}$ computes a function $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$ if there exists a mapping i from $\mathbb{N}^n$ to crystals and a mapping o from crystals to $\mathbb{N}^m$ such that for all inputs $\vec{\alpha} \in \mathbb{N}^n$, $i(\vec{\alpha})$ is a seed crystal such that $\mathbb{S}$ attaches tiles to produce a terminal crystal F and $o(F) = f(\vec{\alpha})$. In other words, if there exists a way to encode inputs as crystals and the system attaches tiles to produce crystals that encode the output. For those systems that allow nondeterministic attachments, the terminal crystal F that encodes the output must contain a special *identifier* tile that we will denote as the ✓ tile.

4.2 3-SAT Tile Assembly

3-SAT is a well-known NP-complete problem. The problem consists of determining whether a Boolean formula in conjunctive normal form (3-CNF) is satisfiable by a truth assignment. The input to the problem is the Boolean formula and the output is 1 if the formula is satisfiable and 0 otherwise.

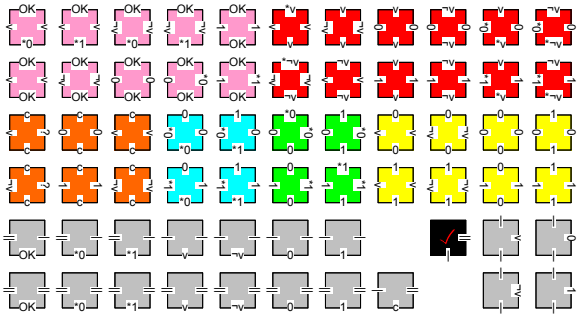

Fig. 1. A tile assembly that solves 3-SAT consists of 64 tile types.

Due to the nature of NP-complete problems, the ability to solve one such problem quickly implies the ability to solve all such problems quickly. For example, if one finds a polynomial-time algorithm to solve 3-SAT, one can then solve the traveling salesman, protein folding, and all other NP problems in polynomial time. Thus, it is sufficient to design a system that uses a large distributed network to solve one NP-complete problem, e.g., 3-SAT, while preserving privacy. We present a tile assembly that solves 3-SAT.

Developing a tile assembly is a process similar to programming or specifying an algorithm. On the surface, tile assemblies are low-level programs, such as instances of Turing machines or cellular automata. However, it is possible to use high-level paradigms, such as encapsulation, abstraction, and recursion to engineer tile assemblies. For example, we have previously designed a multiplication tile assembly [12] that we later use as a subroutine in other assemblies [13]. While developing tile assemblies is a specialized skill, we will describe in Section 5 how a developer can create tile-style software systems without developing new tile assemblies.

Figure 1 shows the 64 possible types of tiles of the 3-SAT-solving assembly. The tiles "communicate" via their side interfaces. Some interfaces contain a 0 or a 1, communicating a single bit to their neighbors. Other interfaces include special symbols such as ∨ and ¬∨ indicating that a variable is being addressed, ⋆ meaning that a comparison should take place, ? meaning the given tile attaches nondeterministically, and | and || indicating the correctness of the computation up to this point. The assembly nondeterministically selects a variable truth assignment and checks if that assignment satisfies the formula. If and only if it does, a special ✓ tile attaches to the assembly.

Every 3-SAT input Boolean formula can be encoded as a sequence of tiles. Such a formula consists of a conjunction of clauses, each of which, in turn, consists of a disjunction of literals. Each literal, either a Boolean variable or its negation, can be encoded with a binary representation of the variable's index and a single bit indicating negation. For example, the literal x_2 can be encoded as three tiles, with labels 1, 0, and ∨, and the

literal $\neg x_3$ can be encoded as three tiles with labels 1, 1, and $\neg$v. We insert a special c tile between the clauses.

Figure 2 shows the progress of the growth of a sample crystal of the tile assembly that solves 3-SAT. The example asks the question whether $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ is satisfiable.

Figure 2a shows the starting seed crystal of the computation. Note that this seed encodes ϕ. For example, the three leftmost tiles of the bottom row encode the literals $10\text{v} = x_2$, $01\neg\text{v} = \neg x_1$, and $00\neg\text{v} = \neg x_0$, which represent the first clause of ϕ. The rightmost column encodes the fact that ϕ contains three variables $10? = x_2$, $01? = x_1$, and $00? = x_0$.

Figure 2b shows the first three tiles (instances of tile types from Figure 1) that attach to the seed. These tiles will make the nondeterministic decision on whether to try $x_0 = TRUE$ or $x_0 = FALSE$. Note that these tiles' left interfaces encode 00v, indicating that the assembly has nondeterministically chosen $x_0 = TRUE$ ($00\neg\text{v}$ would have indicated $x_0 = FALSE$).

Having selected the assignment for x_0, the assembly compares the rightmost literal in ϕ to that assignment. Figure 2c shows that comparison. The top left corner tile with a top interface containing a * indicates that the literal and the assignment match (they are both $00\text{v} = x_0$). If the assignment and literal did not match, the top left corner's top interface would contain no *. Figure 2d shows the comparison of the x_0 assignment to the rest of ϕ. Since the unnegated literal x_0 does not appear anywhere else in ϕ, the rest of the top-row tiles do not contain a * in their top interfaces.

In Figure 2e, the assembly repeats the above steps to nondeterministically select assignments for x_1 and x_2 and compares each of those to the literals in ϕ. Whenever a match occurs, tiles with OK top interfaces propagate that information up, to the top row. Finally, a series of gray tiles attach in the top row to check whether each clause had at least one literal match the assignment. If it does, the special ✓ tile can attach in the top left corner of the crystal. If some clause were not satisfied, no such tile could attach. The fact that Figure 2e contains the ✓ tile indicates that the nondeterministically chosen truth assignment $\langle x_0, x_1, x_2 \rangle = \langle TRUE, FALSE, TRUE \rangle$ satisfies ϕ.

We refer the reader to [15] for the formal proof that this assembly solves 3-SAT. As we explain in Section 5, a single tile assembly, such as the 3-SAT-solving one we have described here, is sufficient to develop tile-style software systems. However, as part of our work on the tile style, we have also provided a tile assembly solution for $SubsetSum$, another well-known NP-complete problem [14]. This second assembly illustrates the flexibility of our work and provides some insight into possible tile-style efficiency improvements.

The tile assembly we have described here follows the algorithm that runs in $O(2^n)$ time. It is possible to leverage more efficient algorithms that solve NP-complete problems to develop efficient tile assemblies. We have already designed one such assembly that solves 3-SAT in $O(1.8394^n)$ time, but do not describe it here because of its complexity. It is important, however, to make clear that tile assemblies can implement the same algorithms used on today's fastest systems that solve NP-complete problems, such as SAT solvers.

5 Tile Architectural Style

Section 4.2 described how tiles can solve a particular NP-complete problem. For each computable function, it is possible to create a tile assembly to compute that function, in an analogous fashion. We have formulated an architectural style [52] that refines a tile assembly into software components and is composed according to the style rules described below. A user may wish to create a custom tile assembly to solve a particular NP problem, but we advise taking an alternate approach. Because NP-complete problems are polynomially related, it is possible to translate all NP problems to 3-SAT using one of the well-established reductions [51]. While we do not focus on the translation process [51], we do note that solving 3-SAT itself has important implications for real-world industrial systems [49]. The time this translation takes is insignificant, as compared to solving either the original or the 3-SAT problem. Thus a user who wishes to solve a particular problem using the tile style needs neither to understand the tile assembly model nor to program with tiles. The user should translate the problem to 3-SAT, or another problem with a known tile solution (e.g., $SubsetSum$ [14]), and use that solution to guide the tile-style architecture. In addition to ease of use, this procedure masks the problem the user is solving. While this is a beneficial side effect of using the tile style, we discuss the much more important privacy-preservation property in Section 7.

The components of the tile-style architecture are instantiations of the tile types of the underlying assembly. A system based on such an architecture will have a large number of components; on the other hand, there is a comparatively smaller number of different *types* of components (e.g., 64 types for solving 3-SAT). Nodes on the network will contain these components, and components that are adjacent in a crystal can recruit other components to attach by sampling nodes until they find one whose interfaces match. Note that many tile components can run on a single physical node, as we will further elaborate below.

In addition to defining the tile types, a tile assembly also directs the architecture how to encode the input to the computation. The input consists of a seed crystal, such as the clear tiles along the right and bottom edges in Figure 2a. Figure 3 summarizes the steps a tile-style system takes to find a solution. During *initialization*, the system sets up a single seed crystal on the network that encodes the input. The seed then *replicates* to create many copies, and each of the copies *recruits* tiles to assemble larger crystals and eventually produce the solution. The

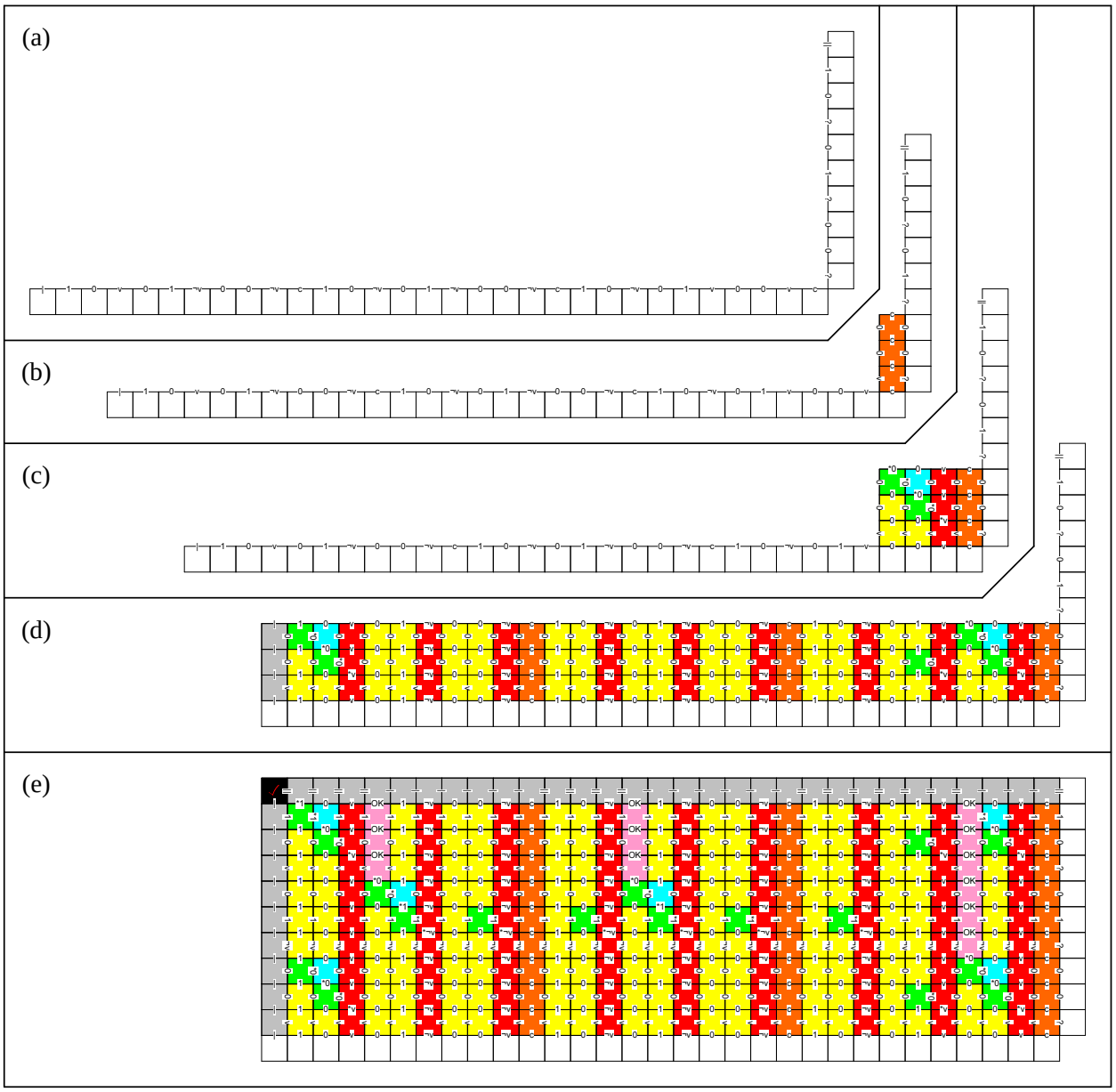

Fig. 2. An example progression of the growth of a crystal of the 3-SAT-solving tile assembly. The crystal seed (a) consists of the clear seed tiles, encoding the input $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$. Specially designed tiles attach to nondeterministically select a variable assignment for x_0 (b); compare that assignment to, first, a single literal in ϕ (c); and then, the rest of the literals in ϕ (d); and, finally, repeat those steps for variables x_1 and x_2 and ensure that each clause is satisfied at least once by the particular selected variable assignment (e). Because ϕ is satisfied when $x_0 = x_2 = TRUE$ and $x_1 = FALSE$, represented by the tiles in the second from the right column in (e), the ✓ tile attaches in the top left corner.

solution tile components (e.g., the ✓ component for the 3-SAT assembly) then report their state to the user. Note that the nodes perform these operations autonomously, without central control, in essence self-assembling the computation.

In order to properly frame the discussion of the four principal tile-style operations in the remainder of this section, we characterize the style using the framework proposed by Mikic-Rakic et al. [45]. Each component's externally visible *structure* comprises its four *interfaces*, shown on the sides of each tile. The *topology* is a 2-D grid of components that allows neighbors on the grid to interact. After the client computer initializes the computation (as described in Section 5.1), the components exhibit four *behaviors*: identifying other nodes on the network that deploy particular types of tiles (as described in Section 5.2), cooperating with neighbors to recruit suitable new components to attach (as described in Section 5.3), replicating to create copies of themselves on other nodes (as described in Section 5.4), and reporting the solution

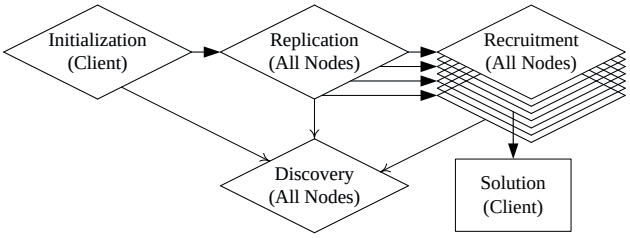

Fig. 3. Overview of tile style node operations.

to the user (as described in Section 5.5). The *interaction* consists of exchanging data about a component's sides in order to recruit. Finally, the *data flow* is limited to the components' interfaces, allowing components to inform their neighbors of their side interfaces, but provide no other information.

5.1 Initializing Computation

The client computer initializes the computation by performing three actions: creating the tile type map, distributing the map and tile type descriptions, and setting up a seed crystal. As our analysis in this section will show, the entire initialization procedure will take on the order of $\log N$ time for a network of N nodes, and each node will send a small amount of data proportional to its local neighborhood size.

5.1.1 Creating the Tile Type Map

A tile type map is a mapping from a large set of numbers (e.g., all 128-bit IP addresses) to tile types. It determines the type of tile components a computer with a given unique identifier (e.g., IP or MAC address) deploys. The tile type map breaks up the set of numbers into k roughly equal-sized regions, where k is the number of types of tiles in the tile assembly. For the 3-SAT example from Section 4.2, there are 64 different tile types, so the tile type map would divide the set of all 128-bit numbers into 64 regions of size 2^{122}. The size of the tile type map, which will later be sent to all the nodes on the network, is small. For an assembly with k tile types, the map is k 128-bit numbers.

For our analysis, we assume that every node on the network is connected to p other nodes, distributed roughly randomly. This is a first-order approximation of the Internet, but our analysis will extend to more accurate models. Every computer may contact its neighbors directly and may query its neighbors for their lists of neighbors. A number of our algorithms are designed specifically to work on such a distributed network, on which no single node knows a large portion of the network. On more highly connected networks, our algorithms can be simplified.

5.1.2 Distributing the Map and Tile Descriptions

The client node distributes the tile type map and a short description of one tile type to a node that deploy that type, as determined by the tile type map. A tile type's description consists of the four tile component interfaces, which can be described using a few bits. The client node contacts at least one node that deploys each tile type by contacting its neighbors, then their neighbors, etc. until at least one node of each type knows the tile type map and its tile type description. The well-known *coupon collector* problem [46] indicates that for a system with k tile types, it will take, with high probability, less than $2k \log k$ time to "collect" a node of each type.

The nodes that learn their types from the client computer propagate the information to their neighbors whose IPs map to the same tile types, and so on, until every computer on the network learns the type of tile component that computer will deploy. Thus every computer receives the tile type map and the description of its own tile type. Each computer might receive its tile type information and the tile type map several times, up to as many times as it has neighbors, which on our network is only p. Each node sends only $\Theta(p)$ data because roughly $\frac{1}{k}$ of a node's p neighbors will have to be sent the $128k$ bits, and $\left(\frac{128kp}{k} = \Theta(p)\right)$. Because the diameter of a network of N nodes with randomly distributed connections is $\Theta(\log N)$ [46], the tile type map and the tile types will propagate through the network in $\Theta(\log N)$ time.

Until now, we have ignored the case of a network with fewer nodes than the number of types of tiles. If the network is that small, it is possible to create multiple virtual nodes on each machine and proceed as before, though a single physical node will have knowledge of more than one tile type, compromising privacy. In the limit, for a network with a single node, it has been analytically shown that privacy preservation is not possible [22].

5.1.3 Creating a Seed

The client is responsible for creating the first seed on the network through a fairly straightforward procedure. For each tile in the seed crystal described by the underlying tile assembly, the client selects a node that deploys that tile type (as we describe in Section 5.2), and asks that node to deploy a tile. The client then informs each deployed tile component who its neighbors on the network are. This procedure is significantly faster and requires significantly less network communication than the distribution of the tile type map.

5.2 Discovery

The node discovery operation is central to the tile style because initialization, replication, and recruitment all use this operation. The discovery operation, given a tile type, returns a *uniformly-random* IP of some computer deploying tile components of that type, meaning that if a node performs this operation repeatedly, the frequencies of the IP addresses it returns asymptotically approach the uniform distribution. Thus, every suitable computer has an equal chance of being returned, in the long run.

Our algorithm for discovery will guarantee uniform-randomness, which in turn will guarantee that all nodes on the network perform a similar amount of computation. The algorithm will use a property of random walks to ensure uniform-randomness.

In order to quickly return the IP address of a computer that deploys tile components of a certain type, each node will keep a table, called the node table, of three IP addresses for each component type, as we explain below. For 3-SAT, the size of this table will be $64 \times 3 = 192$ IPs. The table contains only an identifier for each tile type, and not the details about the interfaces. The preprocessing necessary to create the node table is simple: first a node fills in the table with all its neighbors and then gets help from neighbors (by requesting their neighbor lists). The analysis of this procedure is identical to the analysis of distributing the tile type map; this preprocessing procedure will take $\Theta(k \log k)$ time per node (happening in parallel for each node), for k different tile types. The amount of data sent by each node is limited to $\Theta(k \log k)$ packets. For 3-SAT's $k = 64$, that is fewer than 300 packets, which for typical UDP packets amounts to only 15 kilobytes.

After the preprocessing, when queried for the IP of a computer that deploys tile components of a given type, the node performs two steps: (1) it selects one of the three entries in the node table for that tile type, at random, and (2) it replaces its list of three entries in the table with the selected node's corresponding three entries. The reason for the replacement is that we want the selection of IPs to emulate a random walk on the node graph [46]. The request packet only needs to contain the tile type (e.g., a 32-bit number) and the answer packet must contain three IPs (three 128-bit numbers). This entire procedure takes $\Theta(1)$ time.

We now help clarify the preprocessing and discovery operations with the use of an example. Suppose the network in Figure 4 represents the connectivity of six nodes that all map to the same tile type. In creating its node table, A first checks its neighbors B, C, and D, and records them in the three slots for that tile type. A's node table (for that tile type) is now complete, but had A not found three valid nodes to fill its table, it would expand its neighbor list by querying one of its neighbors for its neighbors, until it discovered a sufficiently large portion of the network. B follows the same procedure as A and creates a node table and records its neighbors A, D, and F as the three nodes deploying the same tile type. When A needs a node of that type later (for reasons discussed below), it selects a random node from its three entries. Suppose it selects B. A then replaces its node table entries with B's entries (A, D, F). Note that it is possible for a node to store itself on its node table.

Lemma 1: On an N-node network, after filling only $\Theta(\log N)$ requests for an IP of a computer that deploys a certain tile type using the above-outlined procedure, the probability of each valid IP being returned is uniformly distributed.

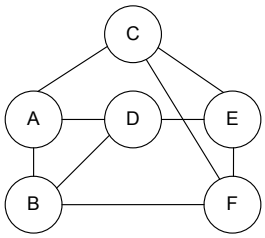

Fig. 4. A network with six nodes. We assume that every node in our underlying network has p neighbors (here $p = 3$).

Proof: Because the node table keeps independent lists of three nodes of each type, it is sufficient to prove the lemma for a single tile component type. Consider the directed graph G formed by representing every node as a vertex with three outgoing edges to the vertices representing the nodes on the node table. Now consider a sequence of nodes derived by the above-outlined procedure of picking a random node from the three entries, and replacing those three entries with that node's entries. That sequence corresponds to a random walk on G. From [46], we know that a random walk on G mixes rapidly, which means that if selecting nodes via this random walk after $\Theta(\log N)$ steps, the probability of getting the IP of each node becomes proportional to that node's in degree. Thus on a uniform graph, every IP is equally likely to be returned. □

We have discussed how to convert a random network into one such that each node has exactly three neighbors. Again we emphasize that this simplification is made to aid our analysis. In fact, the random walk theorem from [46] holds for all graphs with nodes having three or more neighbors, so this result is directly applicable to all reasonable distributed networks. For small networks, discovering the entire network does not pose computational difficulty, and selecting nodes uniformly-randomly is trivial.

5.3 Recruitment

The seed crystal grows into a full assembly by recruiting tile attachments. In a computational tile assembly (such as the assembly described in Section 4.2 that solves 3-SAT), a tile component that has both an upper and a left neighbor recruits a new tile to attach to its upper left. Figure 5 indicates several places in a sample crystal where tile components are ready to recruit new tiles. A recruiting tile component X (highlighted in Figure 5), for each tile type, picks a potential attachment node Y of that type from its node table, as described in Section 5.2, and sends it an attachment request. An attachment request consists of X's upper neighbor's left interface and X's left neighbor's top interface. If those interfaces match Y's right and bottom interfaces, respectively, then Y can attach. At that point, X informs Y of the IPs of its two new neighbors, and those neighbors of Y's IP. Note that X can perform this operation without ever

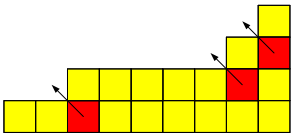

Fig. 5. Tile components that have both an upper and a left neighbor (highlighted in the diagram) can recruit new components to attach to their upper left.

learning its neighbors' interfaces by using Yao's garbled protocol [56], which is crucial for privacy preservation.

Each component's recruitment is a five-step process: X asks N (its upper neighbor) to encode its left interface, N asks W (X's left neighbor) to encode its top interface, W responds to X, X sends attachment requests to a set of potential attachments Y, and those Ys reply to X. We will analyze these five steps in Section 6.5 when we compute the speed of tile-style systems.

In the example from Section 4.2, the successful crystal recruits 310 tile components (non-clear tiles in Figure 2). An unsuccessful crystal, which we discuss further in Section 5.5 can recruit fewer, but no more than 310 tiles.

5.4 Replication

Whenever network nodes have extra cycles they are not using for recruitment, they replicate the seed. Each node X uses its node table, as described in Section 5.2, to find another node Y on the network that deploys the same type components as itself, and sends it a replication request. A replication request consists of up to two IP addresses (two 128-bit numbers) of X's neighbors. X lets its neighbors know that Y is X's replica (by sending Y's IP to X's neighbors). Those neighbors, when they replicate using this exact mechanism, will send their replicas' IPs to Y. Thus, the entire seed replicates. Each component's replication is thus a three-step process: X sends a replication request to Y, Y replies to X, and X tells its neighbors about Y. We will analyze these three steps in Section 6.5 when we compute the speed of tile-style systems.

At the start of the computation, while there are very few recruiting seeds, the replication will create an exponentially growing number of identical seeds (the first seed will replicate to create two, those two will create four, then eight, etc.). When there are sufficiently many seeds to keep the nodes occupied recruiting, replication naturally slows down because recruitment has a higher priority than replication. As some seeds complete recruitment and free up nodes' cycles, replication will once again create more seeds.

The seeds continue to replicate and self-assemble until one of the assemblies finds the solution, at which time the client broadcasts a signal to cease computation by sending a small "STOP" packet to all its neighbors, and they forward that packet to their neighbors, and so on. Since the diameter of a large connected network of N nodes with randomly distributed connections is $\Theta(\log N)$ [46], the "STOP" message will propagate in $\Theta(\log N)$ time.

5.5 Answering 3-SAT in the Negative

A crystal that finds the truth assignment that satisfies the Boolean formula reports the success to the client computer. Since for NP-complete problems the answer is always "yes" or "no," the notification is only a few bits. Deciding that there is no satisfying assignment is more difficult. No crystal can claim to have found the proof that no such assignment exists. Rather, the absence of crystals that have found such an assignment stands to provide some certainty that it does not exist. Because for an input on n variables there are 2^n possible assignments, if 2^n randomly-selected crystals find no suitable assignment, then the client knows there does not exist such an assignment with probability at least $\left(1 - e^{-1}\right)$. After exploring $m \times 2^n$ crystals, the probability grows to at least $(1 - e^{-m})$. Thus as time grows linearly, the probability of error diminishes exponentially. Given the network size and bandwidth, it is possible to determine how long one must wait to get the probability of an error arbitrarily low. In the example from Section 4.2 with 3 variables, the probability of exploring $2^3 = 8$ crystals and not finding the solution is no more than e^{-1}. After exploring 80 crystals, that probability drops to $e^{-10} < 10^{-4}$. Note that no crystal can be larger than 310 tiles, so 80 crystals would require fewer than 25,000 tile components. Because the tile components are lightweight (each one is far smaller than 1 KiB), there is little reason why even a single computer could not deploy that many components.

6 Computational Feasibility

In order to demonstrate that the tile style is a feasible solution for distributing computationally intensive problems on very large networks, we must show that (1) its computational speed is proportional to the size of the underlying network, (2) it is robust to network delay, and (3) real-world-sized problems can be solved on real-world-sized networks in reasonable time.

To that end, we have built Mahjong and Simjong, two software systems that employ the tile architectural style to solve NP-complete problems [11]. Mahjong distributes computation onto computers on a physical network. Simjong is a discrete-event simulator that creates a simulated network of virtual nodes and distributes computation onto that simulated network while employing accurate models of network message delays. In addition to empirically illustrating the above three properties, Mahjong establishes the correctness of our algorithms and demonstrates the distribution of a tile-style system onto a physical network. Because access to Internet-sized networks presents numerous logistical challenges, (the largest physical distributions we could secure are 186 collocated nodes and 100 globally distributed nodes, as part of PlanetLab [48]), Simjong's goal is to accurately

simulate tile-style system distributions on very large networks (e.g., 1,000,000 nodes).

We present the details of Mahjong and Simjong in Section 6.1 and our experimental setup in Section 6.2. We then discuss our experiments testing the scalability, robustness to network delay, and efficiency in Sections 6.3, 6.4, and 6.5, respectively.

6.1 Mahjong and Simjong

Mahjong [11] is a Java-based distributed software system that faithfully implements the tile style and its algorithms described in Section 5. To build Mahjong, we leveraged Prism-MW [44], a middleware platform intended specifically for style-driven implementation of software architectures in highly-distributed and resource-constrained environments. Prism-MW provided explicit implementation-level constructs for declaring components, interfaces, interactions, network communication, etc., as well as the ability to encode stylistic constraints as first-class middleware-level constructs. Each node on a network runs a Prism-MW Architecture, which forms a "sandbox" within which all of the code deployed on a remote node executes. The use of system resources on each participating hardware host is hence restricted and can be released at any time. The tile components deploy inside the Architecture objects and perform their functionality outlined in Section 5 via their interfaces, which are implemented as Prism-MW Ports. Mahjong takes a user-provided description of the set of tiles for an NP-complete problem and the input to the computation (the tiles for solving two NP-complete problems, $SubsetSum$ and 3-SAT, are included) and automates the remaining steps of building a distributed tile-style system. Mahjong consists of 29 objects and 2,900 lines of code. It has proven to be very flexible and robust to variations in the various networks on which we have deployed it to date.

Simjong [11] is a Java-based discrete-event simulator with network-delay simulation capabilities. Simjong executes on a single machine and creates a user-specified number of virtual hardware Node components, each capable of deploying tiles. A central Clock component keeps track of virtual time and allows each Node to execute one instruction per clock cycle. Whenever a Node's tile needs to communicate to another Node's tile, it sends a message via the Network component that determines the delay for that message's delivery. Simjong's network model allows for message delivery time to be constant (e.g., 100ms), chosen at random from some distribution (e.g., Gaussian around 100ms with a 20ms standard deviation), or proportional to the geographic distance between locations assigned to each virtual node (e.g., 50ms between Amsterdam and London, and 500ms between New York and Hong Kong). Simjong's network model is a simplification of the network simulator standard ns-2 [28] because it abstracts away the exact topology of the network, which is not important for our needs.

Simjong's virtual nodes follow the tile-style algorithms, simulating a deployment of Mahjong on a real distributed network. While executing, Simjong keeps track of the number of completed seeds and reports its progress. Thus, it is possible to use Simjong to estimate the time required for a computation to complete after executing only a fraction of that computation. This is an important characteristic that allows us to simulate very large problems on very large networks in comparatively short time.

6.2 Experimental Setup

We leveraged three distributed networks for our experimental evaluation: (1) A private heterogeneous cluster of 11 Pentium 4 1.5GHz nodes with 512MiB of RAM, running Windows XP or 2000. (2) A 186-node subset of USC's Pentium 4 Xeon 3GHz High Performance Computing and Communications (HPCC) cluster [34], whose nodes were distributed in several locations, but all within one city. (3) A 100-node subset of PlanetLab [48], a globally distributed network of machines of varying speeds and resources that were often heavily loaded by several experiments at a time.

The cross-section of data we present in this paper used four representative instances of NP-complete problems, to which we will refer by their labels:

$\mathfrak{A}$: 5-number 21-bit $SubsetSum$ problem,
$\mathfrak{B}$: 11-number 28-bit $SubsetSum$ problem,
$\mathfrak{C}$: 20-variable 20-clause 3-SAT problem, and
$\mathfrak{D}$: 33-variable 100-clause 3-SAT problem.

Our experimental goals were to verify tile style's scalability with respect to network size and robustness to network delay. Our experiments had three independent variables — the number of nodes, the network communication speed between nodes, and the size of the NP-complete problem — and one dependent variable — the time the computation took to complete.

First, to verify the correctness of the tile-style algorithms, we used Mahjong to solve over one hundred of $SubsetSum$ and 3-SAT problems, including $\mathfrak{A}$, $\mathfrak{B}$, and $\mathfrak{C}$. As a rule of thumb, we chose the sizes of instances to each execute in under 4 hours on our 186-node cluster. We verified that, on each of the above three networks, Mahjong found the correct solution to each instance, it sent no unexpected communication between network nodes, and no node produced undesired connections between tile components. Further, we verified that, when provided inputs with a negative answer, Mahjong executed indefinitely, as expected.

We were able to perform experiments with Mahjong on networks of up to 186 nodes and with Simjong on virtual networks of up to 1,000,000 nodes. The need for simulation arose from the difficulty of obtaining time on dedicated machines. Today's distributed-system testbeds are (1) much smaller than the networks the tile style targets, (2) not aimed at computationally intensive (as opposed to data-sharing or threat-prevention)

techniques, and (3) under heavy load. For example, PlanetLab, distributed at 485 locations around the world, consists of only 971 nodes[1], of which almost half are typically unresponsive; of the responsive nodes, some are heavily overloaded or exceedingly slow. Because of these well known issues with PlanetLab [27], we were unable to repeat our experiments as many times as on the other networks. In the end, PlanetLab allowed us to demonstrate that Mahjong can be successfully deployed on globally distributed networks and provided us with useful numerical trends.

6.3 Scalability

To verify that the speed of the computation is proportional to the number of nodes on the underlying network, for each of the three networks described above, we deployed Mahjong on the entire network and on randomly selected halves of the network. We varied the size of the problem and measured the average time in which Mahjong found the solution over 20 executions (except on PlanetLab, as we explained above). We then also deployed Simjong on virtual networks of increasing size from 125,000 to 1,000,000 nodes (with a constant network delay of 100ms for all packets) and simulated the first $10^{-4}\%$ of the seeds to estimate the time required to complete the entire computation. This allowed each Simjong execution to complete in about an hour of actual time, while executing a sufficiently large number of seeds. Our measurements have shown that, after the first few thousand seeds, Mahjong makes fairly constant progress through the seeds and that extrapolating from the $10^{-4}\%$ fraction is accurate.

We hypothesized that as we double the size of the underlying network, Mahjong and Simjong distributions would take approximately half as much time to complete. Figure 6 shows a cross-section of the results of our scalability experiments, which confirm this hypothesis. For example, executing $\mathfrak{D}$ on a 1,000,000-node virtual network took a factor of 1.97 less time than on a 500,000-node virtual network. We speculate that the slight inefficiency on the physical networks (1.9 instead of 2) comes from the constant underlying-network bandwidth; by contrast, increasing the size of a global network is likely to add communication pathways and increase overall bandwidth. The experimental results provide confirmation that the speed of a tile-style system is proportional to the size of the network, resulting in a desirable scaling trend for large networks.

6.4 Robustness to Network Delay

To measure the effect of network delay on the speed of tile-style systems, we compared the times Mahjong took to solve the same problems on equal-sized subsets (up to the maximum 11 nodes) of the private and HPCC clusters and PlanetLab. We then compared the times Simjong took on five virtual networks of 1,000,000 nodes each, with respective network delays of 0ms, 10ms, 100ms, 500ms, drawn from a Gaussian distribution around 100ms with a 20ms standard deviation, and ones proportional to the geographic distance between randomly assigned world-wide locations (varying from 20ms for collocated nodes to 500ms for most distant ones). We again simulated the first $10^{-4}\%$ of the seeds for the same computation on each of those networks to estimate the time required to complete the entire computation.

Network & Problem	Number of Nodes	Execution Time	Speed-up Ratio
Private Cluster	5	43.2 sec.	
$\mathfrak{A}$	10	22.9 sec.	1.89
HPCC	93	220 min.	
$\mathfrak{C}$	186	116 min.	1.90
PlanetLab	50	9.2 min.	
$\mathfrak{B}$	100	4.8 min.	1.92
Simjong	125,000	8.7 hours	
	250,000	4.5 hours	1.93
	500,000	2.1 hours	2.14
$\mathfrak{D}$	1,000,000	64 min.	1.97

Fig. 6. The effect of doubling the network size on the system's execution time. The speed-up ratio is the factor speed improvement over the network of half the size.

We hypothesized that the network delay will have virtually no effect on the time the tile-style systems take to execute. The intuition behind this hypothesis is that each node in a tile-style system handles the deployment of thousands of lightweight tiles and whenever a packet travels between nodes, nodes handle other tiles rather than waiting idly for the network communication to arrive.

Figure 7 shows a cross-section of our empirical data on the physical networks, as well as on virtual networks using Simjong. We found that the execution times were closely clustered and saw no pattern to the small variances (e.g., while the 100ms-network run took more time than the 10ms-network run, the 500ms-network run took less time than the 100ms-network run).

The experimental results are consistent with our hypothesis and provide confirmation that network delay has little to no effect on the speed of a tile-style system. This suggests that tile-style systems can be successful on vastly different networks, from local, highly connected ones to globally distributed, sparse ones.

6.5 Efficiency

The final claim we address in demonstrating the tile style's feasibility for industrial systems is that real-world-sized problems can be solved on real-world-sized networks in reasonable time. In particular, we posit

1. By comparison, DETER [7] users can gain access to only a subset of 300 closely located physical nodes that rely on a network delay simulator similar to the one Simjong uses.

Problem	Number of Nodes	Network Delay	Execution Time
Mahjong			
$\mathfrak{A}$	11	Private Cluster	20.1 sec.
		HPCC	19.3 sec.
		PlanetLab	18.5 sec.
$\mathfrak{B}$	11	Private Cluster	41.6 min.
		HPCC	41.2 min.
		PlanetLab	43.9 min.
Simjong			
$\mathfrak{D}$	1,000,000	0ms	65 min.
		10ms	57 min.
		100ms	64 min.
		500ms	60 min.
		Gaussian	68 min.
		Distance-based	59 min.

Fig. 7. The effect of network delay on system execution time.

that tile-style systems can outperform existing privacy-preserving methods for solving NP-complete problems. There are three ways to solve a highly parallelizable problem while preserving the data privacy: (1) on a large insecure network by using the tile style, (2) on a single private computer, or (3) on a private network of trustworthy computers. We will first discuss the time needed to solve such a problem using the three methods in terms of the number of operations and then discuss the actual time necessary to solve problems.

Suppose a network with N nodes uses the tile style to solve an n-variable m-clause 3-SAT problem. In expectation, the system has to explore 2^n crystals to reach a solution, and each crystal contains $(3m+n)\lg n$ replicated tiles (clear tiles in Figure 2) and no more than $3nm\lg^2 n$ recruited tiles (non-clear tiles in Figure 2). On average, each node will need to replicate $\frac{(3m+n)\lg n}{N}2^n$ tiles and recruit $\frac{3nm\lg^2 n}{N}2^n$ tiles. The replication procedure requires three distinct operations, as described in Section 5.4, each concluded by sending a single network packet; let the time for these operations be denoted as $3i$. Similarly, the recruitment procedure requires five operations, as described in Section 5.3, each also concluded by sending a single network packet; let the time for these operations be denoted as $5u$. Thus, the time required by each node is summarized by Equation (1). This analysis is specific to 3-SAT, but the running times for other NP-complete problems will be very similar, since the fastest-growing factor of 2^n will be the same. (Note that as we discussed in Section 4.2, our analysis here assumes the naïve algorithm that runs in $O(2^n)$ time, but can be extended to more efficient algorithms, such as those used in today's SAT solvers. We use the simple algorithms to allow us to fairly compare tile style-based and traditional systems. However, both types of systems can employ the more efficient algorithms.)

$$\left(3i\frac{(3m+n)\lg n}{N}+5u\frac{3nm\lg^2 n}{N}\right)2^n \tag{1}$$

$$2^n(n+3m)r \tag{2}$$

Now suppose a user wishes to solve the 3-SAT instance on a single computer. That computer would need to examine 2^n possible assignments, and check each n-variable assignment against the m clauses. Equation (2) describes the time this procedure would take using the most efficient available technique, assuming r is the amount of time each operation takes to execute: for each assignment, create a hash set containing the n literal-selection elements and check for each of the $3m$ literals whether the hash set contains that literal. The overhead of using the tile style over a single computer is the ratio of (1) and (2). Assuming $m > n$ and $i = u = r$, meaning that it takes roughly the same amount of time to perform each operation (e.g., looking up a value in a hash set and releasing a message on the network), the ratio is no greater than $\frac{8n\lg^2 n}{N}$. In other words, if the size of the public network exceeds $8n\lg^2 n$, the tile style will execute faster than a single machine. For the sizes of problems we discuss next, that network size is several thousand nodes.

Finally, suppose a user wishes to solve the 3-SAT instance on a private network of M computers. Assuming the best possible distribution of computation and that the network communication is nonblocking, the time this system would require to solve the problem is no less than $\frac{2^n(n+3m)r}{M}$. In this case, the overhead of using the tile style over a private network is $\frac{8n\lg^2 nM}{N}$. In other words, if the size of the public network exceeds $8n\lg^2 nM$, the tile style will execute faster than the private network.

We estimated the time a tile-style system will take to solve a given problem by two methods: (1) empirically determining the values of the constants r, i, and u, and (2) running Simjong. We measured the constants on a 2.4GHz machine running Windows XP and Sun JDK 6.0 by executing several million benchmark tests and averaging their running times. We found that $r \approx 3.6\times10^{-7}$ seconds ($\approx$ 2.8 MHz), $i \approx 2.8\times10^{-7}$ seconds ($\approx$ 3.8 MHz), and $u \approx 4.1\times10^{-7}$ seconds ($\approx$ 2.4 MHz). With these measurements and Equations (1) and (2), we can estimate the speeds of a tile-style system and a single computer solving a given NP-complete problem. For example, solving a 38-variable, 100-clause instance on a single computer would take 3.3×10^7 seconds ≈ 1 year. However, the same problem could be solved using the tile style on a million-node network in 1.8×10^5 seconds ≈ 2.1 days.

Figure 8 compares the execution times of tile-style and single-computer solutions. For each of the three depicted 100-clause 3-SAT instances (with 30, 40, and 50 variables), the graph shows the horizontal line indicating the running time of a single-computer solution, and

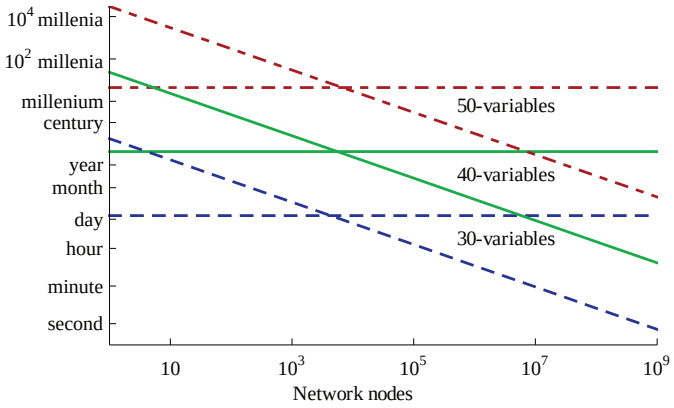

Fig. 8. Expected execution times for single-computer (horizontal lines) and tile-style (diagonal lines) solutions for 30-, 40-, and 50-variable, 100-clause 3-SAT problems.

Number of Nodes	Execution Time	
	Simjong	Equation (1)
125,000	8.7 hours	9.1 hours
250,000	4.5 hours	4.5 hours
500,000	2.1 hours	2.3 hours
1,000,000	64 min	68 min.

Fig. 9. Comparison of execution time for solving $\mathfrak{D}$ as measured by Simjong and estimated by Equation (1).

the diagonal line indicating the running time of a tile-style system implemented in Mahjong and deployed on networks of varying sizes. For networks larger than about 4000 nodes, the tile-style solutions outperform their competitors; for extremely large networks the tile systems are much faster. For example, solving the 40-variable, 100-clause 3-SAT problem on a single computer would take 4 years, while doing so using the tile-style solution implemented in Mahjong and deployed on the network the size of SETI@home (1.8 million nodes [53]) would take 7 days.

To confirm these results, we compared the execution times measured by Simjong with the estimates from Equation (1) for $\mathfrak{D}$. Figure 9 shows the comparison. We have consistently found that Equation (1) was within 8% of the Simjong-measured execution times.

7 Privacy Preservation

We have demonstrated our evidence that tile-style systems are scalable and efficient enough to be able to solve real-world problems. We now argue that tile-style systems exhibit the very property that motives them: privacy of the data. It is difficult to prove privacy preservation empirically because any such analysis can only be as strong as the threat model and its implementation. Instead, in this section, we will analytically argue that, as long as no adversary controls more than half the network, the probability of that adversary learning the input can be made arbitrarily low. This argument provides a strong guarantee of privacy preservation.

To gain intuition into why tile-style systems preserve privacy, consider the information available to a tile-style node. That node deploys several instances of tiles, as in Figure 2. Thus, the node knows about several intermediate bits from several computations, but not of their location in the crystal and of their significance to the computation.

We call a distributed system *privacy preserving* if, with high probability, no node on the network can discover the entire input to the computational problem the system is solving. (We will also discuss, at the end of this section, the probability of discovering parts of the input.) We make two arguments in showing that tile-style systems preserve privacy: (1) given a single tile in a crystal, it is not possible to learn any information about the input and (2) controlling enough computers to learn the entire input is prohibitively hard on a large public network. Proofs and discussion of these statements illustrate that as long as an adversary controls less than half the network, there is an extremely high probability that (s)he cannot learn the input.

1. For a tile assembly, such as the one solving 3-SAT, each tile type encodes no more than one bit of the input. A special tile encodes the solution, but has no knowledge of the input. If every tile component in the crystal were deployed by a different network node, it would be trivial to argue that the computation preserved the privacy. However, since a single node on the network may deploy several tile components of the same type, the argument relies on the fact that each component is unaware of its location in the crystal, and thus does not know the location of the bits of the input. Thus, every node on the network may be aware of either some bits of the input or the solution, but not both, and a node cannot use the partial information it has about the bits of the input to recompose that input in its entirety. That is, the nodes can learn information such as "there is at least one 0 bit in the input," but no more.

2. If an adversary can see the internal data of the entire network, that adversary can learn the input to the problem. However, the likelihood of such a scenario on a very large public network is exceptionally low. An interesting question becomes how much of the network an adversary must control in order to learn the n-bit input.

Lemma 2: Let c be the fraction of the network that an adversary has compromised, let s be the number of seeds deployed during a computation, and let n be the number of bits (tiles) in an input. Then the probability that the compromised computers contain an entire input seed to a tile-style system is $1-(1-c^n)^s$.

Proof: If an adversary controls a c fraction of the network nodes, then for each tile in a seed, the adversary has a probability c of controlling it. Thus for a given n-bit seed, distributed independently on the nodes, the adversary has probability c^n of controlling all the nodes that deploy the tiles in the seed, and thus the probability that the seed is not entirely controlled is $1-c^n$. Since

there are s independent seeds deployed, the probability that none of them are entirely controlled is $(1-c^n)^s$. Finally, the probability that the adversary controls at least one seed is $1-(1-c^n)^s$. □

Let us examine a sample scenario. Suppose we deploy a tile-style system on a network of $2^{20} \approx 1{,}000{,}000$ machines to solve a 38-variable 100-clause 3-SAT problem. Let us also suppose a powerful adversary has gained control of 12.5% of that network. In order to solve this problem, the system will need to deploy no more than 2^{38} seeds, thus the adversary will be able to reconstruct the seed with probability $1-\left(1-2^{-114}\right)^{2^{38}} < 10^{-22}$. Note that as the input size increases, this probability further decreases. The probabilty decays exponentially for all $c < \frac{1}{2}$ (that is, as long as the adversary controls less than one half of the network). In the above example, control of 25% of the network gives the adversary a probability of reconstructing the input below 10^{-11}, and control of 33% of the network yields a probability no greater than 10^{-6}. An adversary who controls exactly half the network has a $\frac{1}{e} \approx 37\%$ chance of learning the input, and one who controls more than half the network is very likely to be able to learn the input, which is why our technique is geared towards large public networks.

One possible challenge to privacy preservation on large public networks is botnets. However, no single botnet comes close to controlling a significant fraction, (say, more than $\frac{1}{1000}$), of the Internet [24]. As the Internet grows, for any fixed-size botnet, the probability that botnet can affect a tile-style system drops exponentially.

We have shown the analysis of the number on nodes necessary to compromise the entire input. The same analysis and exponential probability drop-off applies to reconstructing fractional parts (e.g., one half or one third) of the input. It is somewhat simpler to reconstruct small (e.g., two- or three-bit) fragments of the input, but the information contained in those fragments is greatly limited, can be minimized by using efficient encodings of the data, and for such small fragments, cannot be used to reconstruct larger fragments [20].

Each tile component in the 3-SAT system handles at most a single bit of the input. Theoretically, this is sufficient for solving NP-complete problems; however, practically, handling more than a single bit of data at a time would amortize some of the overhead. Thus each tile component can be made to represent several bits. This transformation would result in a trade-off between privacy preservation and efficiency, as faster computation would reveal larger segments of the input to each node.

8 Contributions

We have developed a new architectural style, called the tile style, as a technique for distributing computation over a network. The tile style provides the opportunity to design large distributed computing systems without having to worry about distribution, privacy preservation, fault and adversary tolerance, and scalability, as those properties are inherent to the style. We presented a rigorous theoretical analysis of the style and formally proved that the resulting systems are efficient and scalable and preserve privacy as long as no adversary controls half of the public network.

We adapted an off-the-shelf middleware platform to create Mahjong, a reusable implementation of the tile style. We deployed Mahjong on several networks, including the globally distributed PlanetLab [48], and simulated its execution on networks of up to 1,000,000 nodes to empirically verify (1) the correctness of tile-style algorithms, (2) that the speed of tile-style computation is proportional to the number of nodes, (3) that network delay has little to no effect on the speed of the computation, and (4) that our mathematical analysis of the time needed to solve large problems on large networks is accurate. For networks larger than about 4000 nodes, the tile style outperforms existing alternatives.

Acknowledgments

We would like to thank Jae young Bang for his help deploying Mahjong on PlanetLab.

References

[1] "Amazon elastic MapReduce," http://aws.amazon.com/elasticmapreduce, 2009.
[2] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID04)*, Pittsburgh, PA, USA, 2004, pp. 4–10.
[3] D. Baker, "Foldit," http://fold.it, 2009.
[4] E. Baldeschwieler, "Yahoo! launches world's largest hadoop production application," http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html, 2008.
[5] A. Balint, M. Henn, and O. Gableske, "A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem," in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT09)*, Swansea, UK, 2009, pp. 284–297.
[6] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.
[7] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab, "Design, deployment, and use of the DETER testbed," in *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, Boston, MA, 2007, pp. 1–8.
[8] B. Berger and T. Leighton, "Protein folding in the hydrophobic-hydrophilic (HP) is NP-complete," in *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB98)*, New York, NY, USA, March 1998, pp. 30–39.
[9] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, 2003.
[10] G. R. Bowman, X. Huang, Y. Yao, J. Sun, G. Carlsson, L. J. Guibas, and V. S. Pande, "Structural insight into RNA hairpin folding intermediates," *Journal of the American Chemical Society*, vol. 130, no. 30, pp. 9676–9678, 2008.
[11] Y. Brun, "Mahjong tile style implementation," http://csse.usc.edu/~ybrun/Mahjong.

[12] ——, "Arithmetic computation in the tile assembly model: Addition and multiplication," *Theoretical Computer Science*, vol. 378, no. 1, pp. 17–31, June 2007.
[13] ——, "Nondeterministic polynomial time factoring in the tile assembly model," *Theoretical Computer Science*, vol. 395, no. 1, pp. 3–23, April 2008.
[14] ——, "Solving NP-complete problems in the tile assembly model," *Theoretical Computer Science*, vol. 395, no. 1, pp. 31–46, April 2008.
[15] ——, "Solving satisfiability in the tile assembly model with a constant-size tileset," *Journal of Algorithms*, vol. 63, no. 4, pp. 151–166, 2008.
[16] Y. Brun, G. Edwards, J. young Bang, and N. Medvidovic, "Online reliability improvement via smart redundancy in systems with faulty and untrusted participants," Center for Software Engineering, University of Southern California, Tech. Rep. USC-CSSE-2009-510, 2009.
[17] Y. Brun and N. Medvidovic, "Fault and adversary tolerance as an emergent property of distributed systems' software architectures," in *Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS07)*, Dubrovnik, Croatia, September 2007, pp. 38–43.
[18] R. Buyya, S. Date, Y. Mizuno-Matsumoto, S. Venugopal, and D. Abramson, "Neuroscience instrumentation and distributed analysis of brain activity data: a case for eScience on global grids," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 15, pp. 1783–1798, 2005.
[19] M. Campbell, A. J. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 57–83, 2002.
[20] M. Chaisson, P. Pevzner, and H. Tang, "Fragment assembly with short reads," *Bioinformatics*, vol. 20, no. 13, pp. 2067–2074, 2004.
[21] A. J. Chakravarti and G. Baumgartner, "The organic grid: Self-organizing computation on a peer-to-peer network," in *Proceedings of the 1st International Conference on Autonomic Computing (ICAC04)*, New York, NY, USA, 2004, pp. 96–103.
[22] A. M. Childs, "Secure assisted quantum computation," *Quantum Information and Computation*, vol. 5, no. 456, 2005.
[23] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[24] D. Dagon, G. Gu, C. Lee, and W. Lee, "A taxonomy of botnet structures," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC07)*, Miami Beach, FL, USA, December 2007, pp. 325–339.
[25] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI04)*, San Francisco, CA, USA, December 2004.
[26] P. T. Devanbu and S. Stubblebine, *Software Engineering for Security: A Roadmap*. ACM Press, 2000, pp. 225–239.
[27] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, and J. Lepreau, "Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems," in *Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets V)*, Irvine, CA, USA, November 2006, pp. 103–108.
[28] S. Floyd and V. Paxson, "Difficulties in simulating the Internet," *IEEE/ACM Transactions on Networking*, vol. 9, no. 4, pp. 392–403, 2001.
[29] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.
[30] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM Symposium on Theory of Computing (STOC09)*, Bethesda, MD, USA, 2009, pp. 169–178.
[31] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *Journal of the ACM*, vol. 38, no. 3, pp. 690–728, 1991.
[32] A. Greenberg, "IBM's blindfolded calculator," *Forbes Magazine*, July 2009.
[33] A. S. Grimshaw, W. A. Wulf, and the Legion team, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.
[34] "High performance computing and communications," http://www.usc.edu/hpcc.
[35] P. Inverardi and A. L. Wolf, "Formal specification and analysis of software architectures using the chemical abstract machine model," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 373–386, 1995.
[36] P. M. Kasson and V. S. Pande, "Control of membrane fusion mechanism by lipid composition: Predictions from ensemble molecular dynamics," *PLoS Computational Biology*, vol. 3, no. 11, p. e220, 2007.
[37] D. Keysers and W. Unge, "Elastic image matching is NP-complete," *Pattern Recognition Letters*, vol. 24, pp. 445–453, January 2003.
[38] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home — massively distributed computing for SETI," *IEEE MultiMedia*, vol. 3, no. 1, pp. 78–83, 1996.
[39] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Proceedings of the Future of Software Engineering (FOSE07)*, Minneapolis, MN, USA, May 2007, pp. 259–268.
[40] O. Kullmann, "New methods for 3-SAT decisions and worst-case analysis," *Theoretical Computer Science*, vol. 223, pp. 1–72, 1999.
[41] M. Lamanna, "The LHC computing grid project at CERN," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 534, no. 1–2, pp. 1–6, 2004.
[42] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande, *Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology*. Horizon Press, 2002.
[43] D. Lucent, V. Vishal, and V. S. Pande, "Protein folding under confinement: A role for solvent," *Proceedings of the National Academy of Sciences*, vol. 104, no. 25, pp. 10 430–10 434, 2007.
[44] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 256–272, 2005.
[45] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic, "Architectural style requirements for self-healing systems," in *Proceedings of 1st Workshop on Self-Healing Systems*, Charleston, SC, USA, November 2002.
[46] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY, USA: Cambridge University Press, 1995.
[47] A. Nakano, R. K. Kalia, P. Vashishta, T. J. Campbell, S. Ogata, F. Shimojo, and S. Saini, "Scalable atomistic simulation algorithms for materials research," *Scientific Programming*, vol. 10, no. 4, pp. 263–270, 2002.
[48] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 59–64, 2003.
[49] S. M. Rubin, *Computer Aids for VLSI Design*. Addison-Wesley, 1994.
[50] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
[51] M. Sipser, *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
[52] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.
[53] Wikipedia, "SETI@home," http://en.wikipedia.org/wiki/SETI@home, 2008.
[54] E. Winfree, "Simulations of computing by self-assembly of DNA," California Institute of Technology, Pasadena, CA, USA, Tech. Rep. CS-TR:1998:22, 1998.
[55] G. J. Woeginger, "Exact algorithms for NP-hard problems: a survey," *Combinatorial Optimization - Eureka, You Shrink!*, pp. 185–207, 2003.
[56] A. C.-C. Yao, "How to generate and exchange secrets," in *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS86)*, Toronto, Canada, October 1986, pp. 162–167.
[57] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user MapReduce clusters," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, April 2009.

Yuriy Brun Yuriy Brun received his Ph.D. degree in 2008 from the University of Southern California and his M.Eng. degree in 2003 from the Massachusetts Institute of Technology. Currently, he is a postdoctoral Computing Innovation Fellow at the University of Washington. His research interests are in the area of engineering self-adaptive systems, and, in particular, using mechanisms from nature to create engineering paradigms for robustness, fault and malice tolerance, scalability, and security. He does (1) theoretical work on design and complexity analysis of biologically inspired algorithms and (2) software engineering work on implementing these algorithms for Internet-sized distributed systems, grids, and clouds. He is a member of ACM and ACM SIGSOFT.

Nenad Medvidovic Nenad Medvidovic received the Ph.D. degree in 1999 from the University of California, Irvine. Currently, he is an associate professor in the Computer Science Department at the University of Southern California. He is a recipient of the US National Science Foundation CAREER award. His research interests are in the area of architecture-based software development. His work focuses on software architecture modeling and analysis; middleware facilities for architectural implementation; product-line architectures; architectural styles; and architecture-level support for software development in distributed, mobile, resource constrained, and embedded computing environments. He is a member of the ACM, ACM SIGSOFT, and IEEE.