Novel Mathematics-Inspired Algorithms for Self-Adaptive Peer-to-Peer Computing *

Steven Y. Ko and Indranil Gupta
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL 61801
{sko,indy}@cs.uiuc.edu

Yookyung Jo
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
ykjo@cs.cornell.edu

Abstract

This paper describes, and evaluates benefits of, a design methodology to translate certain mathematical models into the design of novel, self-adaptive, peer-to-peer (p2p) distributed computing algorithms ("protocols"). This methodology is potentially a good vehicle for translating natural phenomena, representable via mathematical models, into practical p2p protocols. Concretely, our first contribution is a set of techniques to translate certain discrete "sequence equations" rigorously into new p2p protocols called "sequence protocols". Sequence protocols are self-adaptive, scalable, and fault-tolerant, with applicability in p2p settings like Grids. A sequence protocol is a set of probabilistic local and message-passing actions for each process. These actions are translated from terms in a set of source sequence equations. Without having each process simulate the source sequence equations, the emergent behavior of a sequence protocol in a p2p system is equivalent to, and predicted by, its source sequence equations. This paper's second contribution is a new self-adaptive Grid computing protocol called "HoneyAdapt". HoneyAdapt is derived from sequence equations modeling adaptive bee foraging behavior in nature. HoneyAdapt is intended for Grid applications that allow Grid clients, at run-time, a choice of algorithms for executing chunks of the application's dataset. HoneyAdapt tells each Grid client how to adaptively select at run-time, for each chunk it receives, a "good" algorithm for computing the chunk - this selection is based on continuous feedback from other clients. We present analysis, large-scale simulation results, and deployment results.

1. Introduction

The last few years have seen an increased interest in nature-inspired design of self-adaptive algorithms (i.e., "protocols") for distributed peer-to-peer (p2p) systems, e.g., [3, 5, 26]. This paper adopts an alternative approach, that is both rigorous and practical, for translating natural phenomena into novel p2p protocols. Non-Computer scientists have, since the inception of their respective fields, used *mathematical models* to represent their ideas, results, and natural phenomena, e.g., see [21, 22]. Our approach is then to create *design methodologies* [11] that translate these mathematical models into *equivalent* p2p protocols.

Abstractly, a design methodology is targeted at a particular class of mathematical models (e.g., discrete sequence equations in a particular format and structure), and defines a set of techniques that takes as input any instance of the targeted class of models, e.g., a specific set of equations satisfying the specified format and structure. In turn, the methodology generates an output that is a p2p protocol, in the sense that it is an algorithm run completely locally by each process in a decentralized p2p system.

Most importantly, the p2p system-wide behavior of this generated protocol is *equivalent* to that of the set of equations input to the methodology. The protocol running at each process does *not* simulate the sequence equations themselves. Instead, the protocol involves simple local and message-passing actions, so that the *emergent system-wide behavior* is equivalent to the mathematical behavior of the input sequence equations[1].

The concrete contributions of this paper are two-fold:
I. We describe a novel design methodology for translating sequence equations into a class of novel p2p protocols called *sequence protocols*. The techniques in this methodology ensure that the emergent system-wide behavior of a sequence protocol is provably equivalent to, and can be predicted by, the source set of sequence equations from which it is translated. Sequence protocols are completely decentralized, involve low and scalable per-process communication overheads and memory, and are highly fault-

*This work was supported in part by NSF CAREER grant CNS-0448246 and in part by NSF ITR Grant CMS-0427089.

[1]Equivalence will be defined more precisely soon, in Section 2.

tolerant. Most importantly, they are self-adaptive because their emergent behavior in a p2p system inherits the equilibrium points from the sequence equations.

II. In order to demonstrate the utility and properties of nature-based sequence protocols, we design and evaluate a new self-adaptive Grid computing protocol called *HoneyAdapt*. HoneyAdapt is a p2p protocol and is derived from sequence equations modeling adaptive bee foraging behavior in nature. HoneyAdapt is intended for Grid applications that allow Grid clients, at run-time, a choice of algorithms for executing chunks of the application's dataset. HoneyAdapt tells each Grid client how to adaptively select at run-time, for each chunk it receives, a "good" algorithm for computing the chunk - this selection is based on continuous feedback from other clients. We present analysis of the protocol, as well as experimental results from both large-scale simulations involving 1000's of clients, and a smaller-scale 30-node PC cluster deployment.

In general, sequence protocols are intended for large-scale p2p applications that contain hundreds to thousands of processes (e.g., Grid applications, distributed storage, and p2p applications), and that seek to incorporate self-adaptivity by utilizing emergent probabilistic properties of the system. For instance, HoneyAdapt is a generic Grid protocol with applications to parallel sorting, graphics-rendering, astronomy data-processing, etc. For concreteness in this paper, we evaluate HoneyAdapt only in the parallel sorting problem setting (Section 5.2). Further, we briefly present a different sequence protocol called the *Multiplicative protocol* (Section 3).

Related Work: The class of sequence protocols we discuss are a superset of Markov chain algorithms because the latter considers only immediate history of states, while the former considers arbitrarily long histories. Our consideration of sequence protocols is different from the simulation of sequence protocols, as in [23].

Previously in [11], we presented a design methodology to translate *continuous differential* equations into distributed protocols. Unfortunately, those techniques are inapplicable for translating sequence equations as the latter are *discrete*. Further, sequence equations have relatively more pronounced and interesting phase change behavior.

Population protocols [1, 18] also involve large process groups, but are different from ours. Differences in protocol performance for infinite versus finite-but-large group sizes were studied in [15]. Methodologies in general have been used to systematize the design process in many fields, e.g., [4], but in distributed computing, they have begun to emerge only recently, e.g., [16]. Sequence protocols are quite different from checkpointing and virtual synchrony protocols; the latter two are less scalable. Like our basic sequence protocols, many classical distributed algorithms also operate in rounds, e.g., [20]; however, to our knowledge, none are generated from sequence equations.

System Model: For simplicity of analysis, we assume a closed group of N processes, which are non-faulty. The processes communicate by reliable message-passing over a network. N is assumed to be very large. In practice, reliable communication can be provided by TCP channels. The non-faulty process assumption largely holds for Grid settings (e.g., for HoneyAdapt). This assumption can be relaxed in practice for other sequence protocols, e.g., the one in Section 3 works even under massive process failures.

Sequence protocols operate in "rounds" of time, thus we assume all processes know exactly when each round begins. This assumption can be satisfied via one of NTP, TIME, or DAYTIME services (e.g., via NIST servers), that provides coarse granularity synchronization. (Our HoneyAdapt deployment in Section 5.2 does not use the notion of rounds, and is thus asynchronous.) Finally, we assume that each process can sample other processes uniformly at random - this can be implemented via membership protocols such as CYCLON [24] or via a peer-sampling service [14].

Section 2 describes the new design methodology. A case study of a simple sequence protocol is presented in Section 3. Section 4 details the HoneyAdapt protocol, and Section 5 presents experimental results for HoneyAdapt. We conclude in Section 6.

2. Translating Sequence Equations into Sequence Protocols

We present an innovative design methodology that translates certain types of sequence equations into new p2p protocols which we call *sequence protocols*. The canonical single-variable sequence equation is of the form $x_{m+1} = f(x_m, x_{m-1}, \ldots, x_{m-k})$, where k is a constant non-negative integer, f is a function with a finite number of terms, m takes on integer values $\geq k$, and all x_i's take real values $\in [0, 1]$. We also extend our methodology to multi-variable sequence equations with a finite number of variables (say, l variables). This equation system would contain one equation, in the above format, for each of the variables $x1, x2, \ldots xl$, with the right-hand side function for xj denoted as fj. To avoid notational confusion, we will use a subscripted index (e.g., x_i) to represent the time index i for a given variable. We use a side-index (e.g., xj) to denote the j^{th} variable in a multi-variable equation [2]. Thus, xj_i denotes the value of variable xj at time i. Time, in these equations, thus takes discrete positive integer values.

The sequence protocol derived from a system of sequence equations is an algorithm that runs locally at each

[2] We avoid using x^j because superscripts are used in Section 2 to denote power terms. We also avoid $x[j]$ or $xj(i)$ and such expressions, in order to keep our variables uncluttered.

process in the p2p system. Like the original sequence equations, a sequence protocol operates in system-wide *rounds*, where each process knows the exact time at which each round starts. A round corresponds to a discrete time in the sequence equations, i.e., the subscript i in xj_i. At any given time (i.e., at the start or in the middle of a round), state at a process p is maintained locally as an $l - bit$ tuple consisting of l *state variables*. This is denoted as $< x1(p), x2(p), \ldots, xl(p) >$, where the value of the j^{th} bit $xj(p)$ represents whether the process is in (sub-) state xj (if state variable $xj(p)$ is 1) or not. Finally, we use the term xj_i to denote the *system-wide fraction* of processes in (sub-) state xj at time i (for $1 \leq j \leq l, i \in N$). Thus, $xj_i(p)$ is a binary variable ("state variable xj at process p" at time i), while "xj_i" is a system-wide real number $\in [0, 1]$.

Equivalence between the sequence equations and the sequence protocols is defined as follows: for any (i, j), the *value* of xj_i predicted by the sequence equations, is the same as the *fraction* of processes in state xj at time i in the p2p system. Thus, a sequence protocol has the same time-based trajectories and equilibrium points as the source sequence equations. Notice that processes themselves are not simulating the equation, but instead run actions that make the *emergent* behavior in the distributed system the same as the sequence equations.

2.1. Basic Translation Methodology

For the sake of exposition simplicity, let us start with single-variable sequence equations only[3]:

$$x_{m+1} = f(x_m, x_{m-1}, \ldots x_{m-k}) \quad (1)$$

Here, k is a constant non-negative integer, and all $x_i \in [0, 1]$. m is a notation standing for a positive integer representing time. An example sequence equation would be the multiplicative equation $x_{m+1} = r \cdot x_m$.

Suppose f is written as a sum of a finite number of elementary *terms* (positive or negative). Section 2.2 shows that our translation works for multi-variable terms that are either polynomial or non-polynomial. We convert: (i) the equation variable into a state variable for the protocol, and (ii) each term in the function f into a set of protocol actions. For the variable x, a local state variable is defined at each process - this takes on boolean values, indicating whether the process is in that state or not. For process p, we call this as the "state variable x at p". When the state variable at process p is 1, we say that "p is in state x"; when the state variable is 0, we say that "p is out of state x".

The core techniques in the methodology lie in translating the terms from function f into protocol actions. Recall that the protocol actions must ensure that if the *fraction* of processes in state x, at the start of a given round, is the value x_m, and the values of these fractions at the starts of the immediately previous $k - 1$ rounds were $x_{m-1}, \ldots x_{m-k+1}$ respectively, then the fraction of processes in state x at the start of the immediately following round will be equal to: $x_{m+1} = f(x_m, x_{m-1}, \ldots x_{m-k})$.

Overview of Generated Sequence Protocol: First, each process p maintains the following state variables at any time: (i) $x(p)$, the state of process p in the current round, (ii) $x_{-1}(p), x_{-2}(p), \ldots, x_{-k}(p)$, the states of p in the immediately previous k rounds, and (iii) $x_{next}(p)$, a running variable for the state of p for the immediately next round. $x_{next}(p)$ is continuously updated during the current round (based on actions and messages described below). For multi-variable sequence equations, (i)-(iii) is maintained for each of the variables.

Second, at the start of each round at process p, the following initializing actions are taken: (1) for each $i = -k$ to -2, $x_{-i}(p)$ is replaced by $x_{-(i-1)}(p)$, (ii) $x_{-1}(p)$ is set to $x(p)$, (iii) $x(p)$ is set to $x_{next}(p)$, and (iv) $x_{next}(p)$ is initialized to a value of zero. In other words, the oldest remembered state for the process is forgotten and the rest shifted by one, the last round's state is remembered as the most recent state, the current state is updated from the running state variable, which in turn is initialized for the current round.

Once this initialization is complete, process p can respond to messages for this round as well as execute actions for this round. All messages are tagged with the round number in which they are generated, and messages from rounds older than the current one are dropped[4].

In each round, process p executes two types of actions – *Token Generation*, followed by *Token Relay and Apply*. Token generation creates a number of *token* messages, based on the terms in the sequence equations. Token messages are then *relayed* to other processes through random walks. In turn, when p receives tokens, it *applies* these tokens to $x_{next}(p)$. We elaborate below:

1. Token Generation: Each generated token can be either *positive* or *negative*. Positive (resp. negative) tokens are generated for each positive (resp. negative) term in f. The goal of the token generation procedure is to have the N processes create, in a decentralized manner, per round, an expected number of $T \times N$ positive (resp. negative) tokens for each positive (resp. negative) term T, where the value of T is computed by substituting variables within it with fractions of processes in respective states.

2. Token Relay and Apply: When process p, with $x_{next}(p) = 0$, receives a positive token, it *consumes* the token by setting $x_{next}(p)$ to 1. When a process with $x_{next}(p) = 1$ receives a negative token, it consumes the token by setting $x_{next}(p)$ to 0. If none of these two con-

[3]Later (in Section 2.2), we extend our techniques to equations with multiple variables.

[4]This is only a round-based protocol and does *not* require strong underlying mechanisms like virtual synchrony.

ditions apply, the process forwards the received token to a random non-faulty target process. Thus, each token takes a random walk until it is consumed. The following theorem shows the equivalence of the sequence protocol and sequence equations:

Theorem 1: Consider a multi-variable sequence equation system in l variables $\{x1, x2, \ldots, xl\}$ ($xi \in [0, 1]$), and a sequence protocol so that: (i) for each term T appearing in the sequence equation, per round, $T \times N$ tokens are generated and applied as described above (where the value of T in expression $T \times N$ is computed by substituting variables within it with fractions of processes in respective states), with all tokens positive if T is positive and negative otherwise; and (ii) these tokens are relayed and applied appropriately. Then, the sequence equations can be used to predict, at any time (i.e., at the start of any round), the expected fractions of processes that will be in states $\{x1, x2, \ldots, xl\}$, at the start of the next round.

Proof: For each variable index $j \in \{1, 2, \ldots l\}$, the proof follows by induction on (discrete) time. The base induction step is: in the current round m, for each j, let the fraction of processes in state xj be xj_m, and in the immediately preceding k rounds, fractions of processes in state xj were $xj_{m-1}, \ldots xj_{m-k}$ respectively. The induction step is: during the m^{th} round, adding all positive and negative tokens for state xj, the total expected number of tokens for xj is $= N \times fj(\{xh_i\}_{i=m,h=l}^{i=m-k,h=1})$. Due to the relaying, the *probability* of a process having $xj_{next}(p) = 1$ at the end of the m^{th} round, is this number divided by N, or $fj(\{xh_i\}_{i=m,h=l}^{i=m-k,h=1})$, as desired. □

Although a process has to execute token generation at the start of each round, the relay and apply operations are completely asynchronous across processes in the system. Thus, no synchrony is required within the round. Further, in our analysis of the sequence protocols, we will assume that the round duration is long enough for the token relaying to quiesce, i.e., for the system to reach a state where none of the remaining tokens may be consumed anymore by any of the processes. In practice though, this restriction does not violate our scalability goals. Each token would typically transit through a logarithmic number of processes until it is consumed. Thus, in a given round, if each process generates a constant number of tokens, the message overhead per process will be logarithmic in group size. Finally, our protocol implementations limit the length of a round - thus extra tokens can be lost if the round expires - yet, our experiments in Sections 3 and 5 show this does not affect convergence behavior.

2.2. Token Generation

At the start of the round, each process p generates a series of tokens *for each term in each source sequence equation*, based on the following rules. For Theorem 1 to hold, we desire that for each term T in the sequence equations, an expected $T \times N$ tokens are generated system-wide per round. We describe below how polynomial, multi-variable, and non-polynomial terms, all fall under Theorem 1.

2.2.1 Polynomial Terms:

Constant term of form $T = r$, where r is a constant: Process p (regardless of its current state) *generates an average of $|r|$ tokens*. This is achieved by generating $\lfloor |r| \rfloor$ tokens, and then generating an extra token with probability $(|r| - \lfloor |r| \rfloor)$. If $r > 0$, all these tokens are positive, otherwise they are all negative. Since each processes creates an expected r tokens, the expected number of tokens created system-wide is $r \times N$. Notice that this token generation action requires no message exchange.

Linear terms of form $T = r \cdot x_{m-j}$, where $j \leq k$: Process p first checks if j rounds ago, it was in state x. If indeed $x_{-j}(p) = 1$, then p generates an average of r tokens, in a similar manner to the constant term translation in the last paragraph. If $x_{-j}(p) = 0$, then p generates no tokens. If the current round is m, then a fraction x_{m-j} processes satisfy $x_{-j}(p) = 1$, thus the expected number of tokens generated is $x_{m-j} \times N \times r = T \times N$. Notice that this token generation action requires no message exchange.

Multiplicative terms of form $T = r \cdot \Pi_{i=m}^{i=m-k} x_i^{(j_i)}$ (each j_i a non-negative integer, some j_i positive): Let i' be the highest value of i in the term T such that exponent $j_i > 0$. Process p first checks if $(m - i')$ rounds ago it was in state x, i.e., if $x_{-(m-i')}(p) = 1$ (where $x_0(p) = x(p)$). If not, it takes no action. If yes, it sends out $(\sum_{i=m}^{i=m-k} j_i) - 1$ sampling messages, each to one target process, chosen uniformly at random. Target processes acknowledge the sampling message - for each sampling message, random targets are retried (after a time-out) until an acknowledgement is received. Non-faulty target processes reply immediately with the list of states they were in for the last k rounds. Process p then checks whether for all $b = 1$ to $(\sum_{i=m}^{i=m-k} j_i) - 1$, the b^{th} process that sent a reply was in the state indicated by the b^{th} variable occurring in the product $T/r \cdot x_{i'} = x_{i'}^{(j_{i'}-1)} \cdot \Pi_{i=i'-1}^{i=m-k} x_i^{(j_i)}$, when the individual variables of the product are arranged arbitrarily. If this condition is true, process p generates an average of r tokens, otherwise it generates none. Since the probability of this condition being true at a process is $T/r \cdot x_{i'}$, and the fraction of processes with $x_{-(m-i')}(p) = 1$ is $x_{i'}$, the expected number of tokens generated system-wide is $x_{i'} \times T/r \cdot x_{i'} \times N = T \times N$.

2.2.2 Translating Sequence Equations with Multiple Variables:

The presented methodology (with techniques from both Sections 2.2.1 and 2.2.3) can be generalized to translate a system of sequence equations. A system of se-

quence equations with $l > 1$ variables $\{x1, x2, \ldots, xl\}$ specifies a sequence equation for each $xj (1 \leq j \leq l)$: $xj_{m+1} = fj(xh_b, 1 \leq h \leq l, m-k \leq b \leq m)$. Here, fj has a finite number of terms, k is a constant non-negative integer, and $\forall h, b : xh_b \in [0, 1]$. In the derived sequence protocol, for each j $(1 \leq j \leq l)$, each process maintains one state variable xj, along with a memory of the last k values of it. Separate token generation actions are created from each fj and executed at each round. The tokens generated from fj are tagged with j so that they are applied only to state variables xj. Note that if fj contains terms with xh_* $(h \neq j)$, then the token generation actions related to state variable xj easily generalize may involve sampling values of state variables other than xj.

2.2.3 Non-Polynomial Terms: Translation of a non-polynomial term requires each round to be split into a known number of *subrounds*, and the use of *subtoken* messages. Just like for rounds, each process knows when each subround starts. Subtokens do not affect state variables directly, but instead contribute towards generation of tokens. The actions for individual terms are tied together by calculating the maximum number of subrounds across all terms, and using this as the number of subrounds per round in the overall sequence protocol. We ignore such tying issues below, focusing only on individual terms.

Division terms $T = r/x_{m-i}$ $(0 \leq i \leq k$, with $x_0 \equiv x)$: Split each round into two subrounds. In the first subround, process p (regardless of its state at the round start) iteratively selects one target process uniformly at random (which acknowledges with the value of its local $x_{-i}(p)$), until a reply is received from a target process that satisfies $x_{-i}(p) = 1$. A count of the number of acknowledging targets is kept. If their total is z, p generates an average $r \cdot z$ tokens. In the second subround, the tokens are relayed and applied as usual. Since $E[z] = 1/x_m$, the expected number of system-wide tokens is $N \times r \cdot 1/x_m$, as desired.

Fractional Terms: In a sequence equation with l distinct variables $x1, x2, \ldots, xl$, consider a canonical term $T = r \cdot \frac{\sum_{j=1}^{j=l} b_j . a_j . xj}{\sum_{j=1}^{j=l} a_j . xj}$, where a_j's are all positive real numbers, and each b_j is a boolean with value either 0 or 1.

To translate this term, two subrounds are required per round. In the first subround at process p, for each j such that $xj(p) = 1$, p generates an average a_j subtokens. Each subtoken is tagged with the value of j that generated it. Then, p multicasts these subtokens to *all* other processes in the group. In the second subround, p selects one subtoken at random from those received. Suppose the tagged value is j' - if $b_{j'}$ (in term T) is 1, process p generates an average r tokens; if $b_{j'}$ is 0, p generates no tokens[5].

[5]If a process receives no subtokens after the first subround, it needs to sample other processes to "copy over" their received subtokens.

Each process receives, via the multicast, for each j, $N \times a_j \cdot xj$ subtokens carrying j. The above sampling then means that it generates r tokens only with probability T/r. Thus the expected number of tokens generated system-wide is $N \times T/r \times r = T \times N$.

Recursive Translation: Define a term T as *translatable* if any of the above techniques (including this one) can be applied to it, so that the derived protocol generates an expected $T \times N$ tokens per round. Now, given a term T', suppose it can (1) be rewritten into one of the above forms (polynomial, division, or fractional) by substituting with dummy variables a finite set of *sub-terms* occurring in it, so that: (2) each of the sub-terms itself is translatable. Then T itself is translatable. This recursive translation can be achieved by designing actions for term T' as follows: split the round into two sub-rounds, and translate each substituted sub-term (say g) in the first sub-round by generating subtokens and relaying these subtokens until each process has either 0 or 1 subtokens for each substituted sub-term. The second subround uses the translated action from T, except that the actions relating to the substituted sub-term sample whether or not a process received a g subtoken.

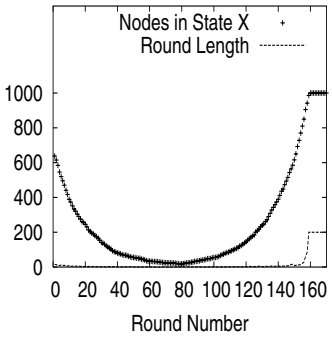
3. Case Study: Detecting Global Triggers

In order to demonstrate that even the simplest sequence equation can lead to useful sequence protocols, we present a case study of the *Multiplicative protocol*. This is derived from a one-variable sequence equation containing only one term, yet can be used to detect a threshold-based global property. The reader will find sequence protocols for detecting other global properties in [12].

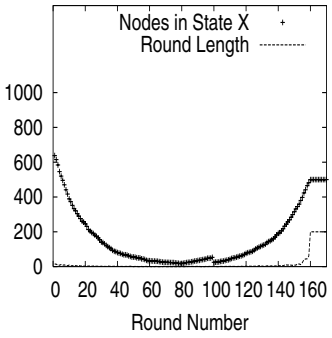
In a self-adaptive p2p system, we are interested in a global *threshold trigger* for a metric of interest, denoted as R. Concretely, suppose each process p proposes its value R_p for R. Then we wish to detect whether the system-wide average value of R_p is above or below a specified threshold $thresh$. A threshold trigger can be used to detect, for instance, possible partitioning of a p2p overlay, by setting R_p = number of neighbors of process p, and $thresh$ to a value based on the overlay structure (e.g., for a random overlay, $thresh = O(\log(N))$). This problem is solved by the Multiplicative protocol, described below.

The Multiplicative Protocol: The Multiplicative protocol is derived from the multiplicative equation $x_{m+1} = r \cdot x_m$ $(r > 0$ and $x_* \in [0, 1])$. The sequence protocol is derived using the polynomial linear translation (Section 2.2.1). The resulting sequence protocol has only one action per round per process.

The emergent behavior of the protocol can be analyzed via the sequence equation itself. Equilibrium occurs when $x_{m+1} = x_m$, i.e., $x = r \cdot x$, or $x \cdot (1 - r) = 0$. For all $r \neq 1$, the only equilibrium point is $x = 0$. For stability of equilibria, we use a well-known result from Lorenz [22]:



(a)



(b)

Figure 1. Multiplicative Protocol: Basic Phase change and Massive Failure. (a) The value of r_{avg} changes from 0.95 to 1.05 at t=80. $N = 1000$. (b) Same as (a), except that 50% of the processes fail at t=100. Convergence occurs at t=206. $N = 1000$.

Lorenz's Stability Condition: For a sequence equation $x_{m+1} = f(x_m)$, an equilibrium point $x = x_\infty$ is stable if and only if $|f'(x)|_{x=x_\infty} < 1$.

For the multiplicative map, $f'(x) = r$ at all x. If $r < 1$, the system converges towards the sole stable point $x = 0$. However, if $r > 1$, $x = 1$ is the sole stable point while $x = 0$ becomes an unstable point.

Convergence Times: We are interested in X, the number of processes in state x. For $r < 1$, with $X = N$ initially, $E[X]$ after t rounds is $= (N \cdot r^t)$. The expected convergence time towards $X = 0$ is $\log_{1/r}(N)$ rounds. For $r > 1$, $X = 1$ initially implies that after t rounds $E[X] = r^t$, giving an expected convergence time of $\log_r(N)$, towards $X = N$.

Average Threshold Trigger Detection: To implement a threshold trigger $thresh$ for metric R, we use the Multiplicative protocol where each process p uses a value of $r = \frac{R_p}{thres}$ for the sequence protocol. In the multiplicative equation, replacing r with $r_{avg} = \sum_p(r_p)/N$, does not change the behavior of the equation. Thus, when $r_{avg} < 1$, the systems stabilizes with all processes out of state x; when $r_{avg} > 1$, the system converges and stabilizes with all processes in state x. Thus, detection can be achieved at an initiating process by querying, after enough time for convergence, the current states of the a small *control group* of processes selected randomly. If a majority of processes in the control group are found to be in state x, then the initiator concludes that the system-wide average of R_p is $> thresh$, otherwise it concludes that this average is $< thresh$.

Experimental Results: We give simulation data from a synchronous simulation done on a PC with 1.7 Ghz Intel Celeron CPU, 256 MB RAM, WinXP Pro. The Mersenne Twister pseudorandom generator was used. There are $N = 1000$ non-faulty processes. Round length is limited to 200 periods, each process receives and sends tokens once per period, and the overlay is a complete graph. Figure 1 plots both the number of processes in state x, as well as the time until the last token is consumed in each round. Figure 1(a) shows that when r_{avg} is changed from 0.95 to 1.05 at t=80 onwards, the system dramatically changes from a convergence on $x = 0$ to a convergence on $x = 1$. Also, when $r_{avg} > 1$, not all tokens are consumed since each round becomes the maximum 200 periods, but this does not affect convergence. Figures 1(b) shows that massive failure of 50% of processes (at time $t = 100$) does not affect convergence.

4. HoneyAdapt: Adaptive Grid Computing

We present HoneyAdapt, a self-adaptive protocol for Grid computing. HoneyAdapt is generically applicable in several highly parallelizable Grid applications that use the master-worker paradigm, e.g., SETI@Home, physics and meteorological applications, etc. In the master-worker paradigm, a *master host* initially contains the (large) data set that has to be processed. The master then hands out data chunks to other *clients* (workers), connected amongst each other through a p2p overlay. All chunks are assumed to be the same size. Each client gets one chunk at a time.

The problem we address is the following. The clients are free to use any one of several alternative algorithms $1, 2, \ldots, A$ for each chunk. However, neither master nor the clients can guess which algorithm is the "best" for any chunk in any given data set. This is especially true of data sets with mixed types of chunks, where individual chunks may have a different "best" algorithms. This goal is similar to that of AI algorithms for evaluation and tracking of "experts" [13]. However, those are exhaustive and potentially expensive if applied to a p2p distributed system.

Our solution to this problem is called *HoneyAdapt*. HoneyAdapt is a sequence protocol derived from sequence equations that model the foraging behavior of honeybees as they attempt to select, in a decentralized manner, the "best" nectar source(s) from among multiple available sources [9, 21]. HoneyAdapt allows clients to adapt among A component algorithms *on the fly*, i.e., at run time. The most attractive aspect of HoneyAdapt is its *black-box* nature; HoneyAdapt requires no knowledge of the internal characteristics of the application or input data, instead using active

feedback from the clients at *run time*. However, as our experiments indicate, HoneyAdapt works best when long sequences of chunks in the data-set are "similar" in nature.

We describe the protocol in Section 4.1 and its analysis in Section 4.2. HoneyAdapt is applicable to parallel sorting, graphics rendering, and other data processing applications. We show in Section 5.2 that when applied to parallel sorting, HoneyAdapt (called *HoneySort*) *outperforms traditional sorting algorithms such as Quicksort and Insertion sort.* HoneyAdapt and HoneySort are proof of concept systems, and are not (yet) meant to compete with state of the art sorting systems [2]. HoneyAdapt can also be easily optimized for specific applications, e.g., by combining it with machine-learning techniques; however, these are beyond the scope of this paper.

4.1. The HoneyAdapt Sequence Protocol

We describe the HoneyAdapt sequence protocol below. This sequence protocol runs synchronously in rounds. During each round, each client processes only one chunk. For clarity, the initial description also refers to the sequence equation and the honeybee behavior. Consider a group of clients (honeybees) trying to select the best from among A multiple algorithms (nectar sources) based on measured individual running times for the chunks at the clients (measured quality of nectar sources). At any point of time, a given client (honeybee) is said to be in state i if it is using algorithm i on the current chunk (if honeybee has just foraged, i.e., visited, nectar source i), where $i \in \{1, 2, \ldots A\}$. Let the fraction of clients (honeybees) in state i at start of round t be $a_i(t)$. We use a variant of the equations in [9] that model the actual honeybee behavior:

$$a_i(t) = a_i(t-1) \cdot (1 - pf_i) + \left(\frac{sq_i \cdot pd_j \cdot a_i(t-1)}{\sum_{j=1}^{A} sq_j \cdot pd_j \cdot a_j(t-1)} \right) \cdot \sum_{j=1}^{A} pf_j \cdot a_j(t-1) \quad (2)$$

This is an $A-$variable system of sequence equations, and HoneyAdapt is derived from it using techniques for fractional, polynomial, and recursive translation from Section 2. Although the sequence protocol would have each state maintain an $A-$tuple of states (one for each variable), we have the restriction that each client can be using exactly one algorithm at any given point of time, i.e., at any client, exactly one of the A state variables is 1. This means that at any time t, $\sum_{i=1}^{A} a_i(t) = 1$. In addition, our initial conditions are that for each i, at least one client uses algorithm i, i.e., for each i, some client p has $a_i(0)(p) = 1$ initially.

In the above equation, for each i, sq_i is the *quality* of algorithm i (nectar source i), calculated on the fly depending on the algorithm and the chunks currently being processed. Also, pf_i is the *following probability*, and pd_i is *dancing probability* - both $\in (0, 1]$ and are fixed.

When a client uses algorithm $i \in \{1, 2, \ldots A\}$, on a data chunk, it calculates a "quality" for i, denoted as $sq_{i,p}$. $sq_{i,p}$ is inversely related to the running time for that chunk; a shorter execution time translates to a higher quality. sq_i in equation (2) is thus the average of $sq_{i,p}$ across all clients p using algorithm i in that round. At the round's end, p decides to *dance* with dancing probability pd_i (similar to a bee dance to advertise the nectar source it just foraged). Dancing means generation of a number of *advertisement messages supporting algorithm* i. The number of advertisements generated at client p is proportional to quality $sq_{i,p}$ in the round. These are then multicast to *all* other clients.

Then, with following probability pf_i, client p decides to *follow*. This means *p samples a random advertisement* from those received; the value of j supported therein will be used as next round's algorithm (this accounts for the second term on the right hand side of equation (2)). On the other hand, if p decides not to follow (with prob. $1 - pf_i$), it continues to use old algorithm i (this accounts for the first term on the right hand side of equation (2)).

4.2. Analysis of the HoneyAdapt Protocol

Theorem 1 tells us that HoneyAdapt's emergent behavior is equivalent to the equation (2). If for all $i = 1$ to A, the fraction of processes using algorithm i in round $(t-1)$ is $a_i(t-1)$, then the fraction of processes using algorithm i (for any i) in round t is that specified by the equation (2). We analyze below the convergence behavior of HoneyAdapt.

Lemma 1: If all sq_i's are unique and non-zero, all pd_i's are equal to each other, and all pf_i's are equal to each other, then HoneyAdapt has exactly A equilibrium points, each corresponding to a unique i with $a_i(t) = 1$.

Proof: $\forall i$, let $pd_i = pd$ and $pf_i = pf$. Equilibrium points occur when $\forall i : a_i(t) = a_i(t-1)$, i.e., from equation (2), when $\forall i$: either $a_i(t) = a_i(t-1) = 0$, or $\frac{pf}{sq_i \cdot pd} = \frac{\sum_{j=1}^{A} pf \cdot a_j(t-1)}{\sum_{j=1}^{A} sq_j \cdot pd \cdot a_j(t-1)}$. Notice that the right hand side of the second equation is independent of i. There cannot be any equilibrium point with stable values $a_i \neq 0, a_j \neq 0, i \neq j$, since then $sq_i = sq_j$, a contradiction. Hence proved. □

Theorem 2: If all sq_i's are unique and non-zero, all pd_i's are equal to each other, and all pf_i's are equal to each other, then HoneyAdapt has only one *stable* equilibrium point, at which $a_{i_{max}} = 1$, where $\forall j \neq i_{max} : sq_{i_{max}} > sq_j$.

Proof: An equilibrium is stable iff all eigenvalues of the Jacobian matrix have absolute values < 1. Without loss of generality, let $i_{max} = A$. Since $a_A = 1 - \sum_{i=1}^{A-1} a_i$, we need to consider only the $(m-1) \times (m-1)$ Jacobian matrix H. This turns out to be:

$$H_{ij} = 0\,(i \neq j)\,, H_{ii} = 1 - pf + \frac{sq_i}{sq_A} \cdot pf$$

Here, $1 \leq i, j \leq m-1$. Now, we use an easy extension of Lorenz's stability condition to multi-variable sequence equations (for a proof of this extension, see [12]): in a sequence equation with multiple variables, the stability con-

dition at an equilibrium point is that all eigenvalues of the Jacobian at that point should have absolute values < 1.

The eigenvalues of the above matrix are its diagonal elements. Since $pf \in (0, 1]$, we have $H_{ii} \geq \frac{sq_i}{sq_A} \cdot pf > 0$. Recall that $\forall i : 1 \leq i < A$, we have $sq_i < sq_A$. Thus, all eigenvalues $H_{ii} = 1 - pf \cdot (1 - \frac{sq_i}{sq_A}) < 1$. Hence, absolute values of all eigenvalues are < 1. □

Convergence Times: Consider a small perturbation from the equilibrium point $a_A = 1$. From Theorem 2, we have that $\forall i \neq A$: $a_i(t) = \left(1 - pf\left(1 - \frac{sq_i}{sq_A}\right)\right)^t a_i(0)$. The convergence speed time is thus logarithmic in N. Notice that the message traffic depends on parameter pd, but pd does not affect equilibria or convergence speed.

5. Experiments with HoneyAdapt

Section 5.1 presents simulations of HoneyAdapt, and Section 5.2 its behavior for the parallel sorting problem.

5.1. HoneyAdapt Simulation Results

We experimentally study HoneyAdapt and several variants of it, via our implementation in a custom C simulator. Our implementation relaxes many of our previously stated assumptions. Firstly, each client (node) at the end of a chunk execution, does not create tokens as in Section 4.1. Instead it creates a *single advertisement message* carrying a weight that is set to the quality of the used algorithm on the latest chunk. Secondly, HoneyAdapt as described so far, had each client choose its current chunk's algorithm either by sampling advertisements ("sample") or by retaining its recently-used algorithm ("memory"). However, we evaluate an additional three strategies defined by {memory vs. memoryless} × {greedy vs. sampled}.

We use a network topology generated via BRITE's [17] flat router Waxman model, with 1000 routers. Each client is assigned to one random stub in the network topology. The Grid computing overlay topology is a random graph, with each client selecting $M = 6$ random neighbors[6].

Table 1 summarizes our experimental parameters. We consider a master-client Grid application, where the master data is split into chunks that are handed out sequentially to requesting clients. Each client has a choice of a fixed set of algorithms to process each fetched chunk, and uses HoneyAdapt to make this run-time choice. HoneyAdapt's success depends on series of chunks being "similar" in nature - we study this by assigning each chunk one of several discrete but finite *types*. Types differ in that each has a different set of average execution times for each of the component algorithms, given a chunk of that type. For each type, we first randomly rank all algorithms, i.e., from the fastest to slowest. Next, we assign a per-chunk average execution time for each algorithm, according to its $rank$, as $(min\_duration + rank \cdot distance)$ seconds. At runtime, the execution time of each chunk of a particular type is chosen independently, and as a random number in an interval $[-range, +range]$ around the above average. Finally, types are assigned to chunks using a parameter *cluster size*, where a cluster is a sequence of consecutive chunks of the same type. Each cluster's type is selected randomly.

[6]Higher values of M gave similar results, hence are not shown.

Table 1. Default Parameters for Simulations.

# of total chunks	100,000
Cluster size	10,000
# of types	10
# of algorithm	10
min_duration	10 seconds
distance	10 seconds
range	1 second
dancing probability	1.0
following probability	0.9
propagation	randomwalk with 10 hops
# of nodes	1,000

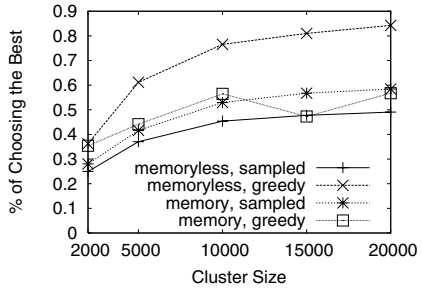
Our main metrics are: (1) % chunks that end up being executed with their individual fastest component algorithms, and (2) running time for the entire data set, calculated as sum of execution times for all chunks across all clients.

Algorithm Variations: We evaluate four variants of HoneyAdapt, as defined by {memory vs. memoryless} × {greedy vs. sampled}. Recall that the original algorithm in Section 4.1 uses {memory, sampling}. In comparison, *memoryless* means that a client, that decides not to follow, chooses a next-algorithm *at random* from among all choices. *Greedy* means that a client, that decides to follow, selects as the next-algorithm *greedily*, by choosing an algorithm with the highest quality among all received advertisement messages within the last round only.

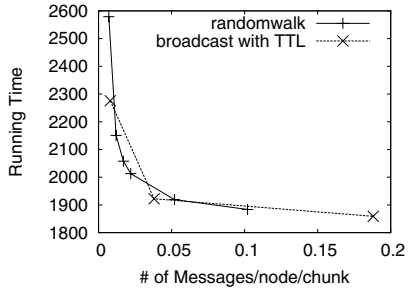
Figure 2(a) shows the percentage of choosing the best algorithm for a set of chunks. When *memoryless* strategy is used, there is a significant difference between *greedy* and *sampled*, while using the *memory* strategy shows no difference between *greedy* and *sampled*. Since {*memoryless, greedy*} strategy gives the best performance, we use it as a default in the rest of our experiments.

Propagation Methods: Figure 2(b) shows a comparison of two techniques to spread advertisement messages in the overlay - random walk (with a TTL=time-to-live) and a broadcast (with a TTL). Different plot points correspond to different values of TTL. The two algorithms define similar boundaries, hence we use a random walk from now on.

Cluster Size: Figure 3 varies the number of consecutive chunks of same type (cluster size) and plots the percentage difference between the actual running time and the optimal running time for the data set. The optimal running time is a lower bound on the runtime, since it is the total execution time assuming *each chunk* is executed with the best (i.e.,



(a)



(b)

Figure 2. HoneyAdapt Variants. (a) Algorithm selection: {memory vs. memoryless} $\times$ {greedy vs. sampled}. (b) Message propagation: randomwalk vs. broadcast with TTL

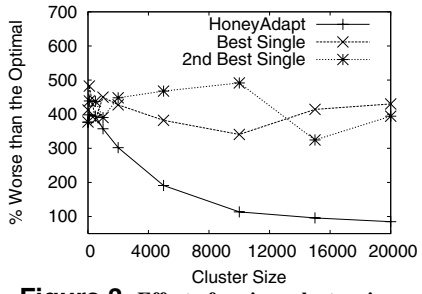


Figure 3. Effect of various cluster sizes.

fastest) algorithm for it[7]. For 1000 nodes in the system, notice that (1) HoneyAdapt does far better than using only one algorithm for all the chunks in the data set (we plot the best two algorithms for the data set), and (2) the performance of HoneyAdapt levels out after a cluster size. This indicates that with 1000 nodes, for cluster sizes over 10,000, HoneyAdapt is only about twice as worse as the optimal.

Scalability: Figure 4 shows the effect of varying the number of nodes from N =200 to 4,000. Cluster size is set as $10 \cdot N$, in order to allow nodes to adapt sufficiently well. In order to maintain a constant per-node bandwidth due to advertisements, these are spread using a random walk with TTL scaled up with N as $TTL = \frac{N}{500}$. The plot shows that the bandwidth and relative running time of HoneyAdapt both improve with N. Notice that the bandwidth is high for

[7]Note that this optimal variant is difficult to implement in a blackbox manner, i.e., without knowing the nature of chunks.

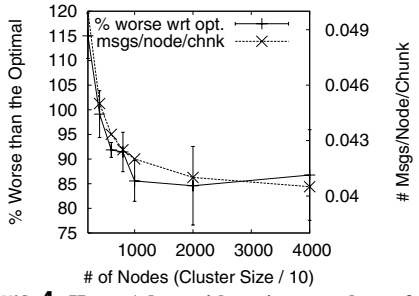


Figure 4. HoneyAdapt with various numbers of nodes.

low values of N because of the non-scalable cost at each node, of fetching each chunk from the master.

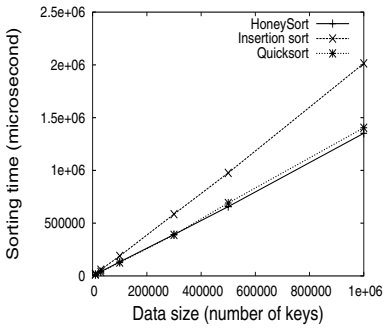
5.2. HoneySort Protocol and Experiments

We present HoneySort, HoneyAdapt's variant for the parallel sorting problem[8]. In this problem, the master initially partitions the input array into approximately equal sized chunks by using multiple pivots, so that it can concatenate the final results from all clients to produce the output. Each client has a choice of several sorting algorithms for each chunk. HoneySort differs from HoneyAdapt in three aspects: (1) It is *completely asynchronous and does not use the concept of rounds at all*. Thus, when a client p sorts a chunk, it fetches the next chunk immediately. (2) It uses an additional probabilistic parameter called randomization probability, denoted as pr. While picking an algorithm for the next chunk, a client chooses a random algorithm with probability pr, otherwise it uses the default following algorithm with pf ({memory, sampled}). (3) Advertisements are forwarded only to immediate overlay neighbors, and only the latest advertisement from each neighbor is used.

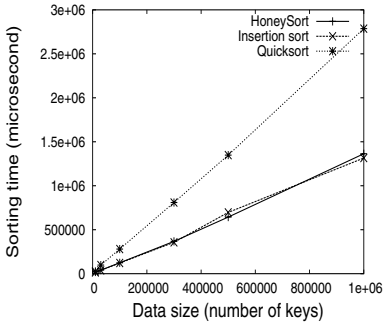
We deployed HoneySort on a Linux PC cluster with a TCP-based overlay, which is a complete graph by default (see Figure 7 for random overlays). Component algorithms are classical Quicksort and classical Insertion sort [7]; randomized quicksort and mergesort yield similar results. Datasets consist of 8 B integer element input arrays created via SortGen [10].

Figures 5(a) and (b) show running times respectively for completely random and pre-sorted input arrays. In each case, HoneySort is as fast as the best algorithm - Quicksort and Insertion sort respectively. Figure 6(a) shows data for heterogeneous arrays, where groups of *dynamic change unit* chunks are selected within the input array, and groups are alternately random and sorted. Thus, an x-axis value of 20 means the first 20 chunks are sorted, next 20 randomized, next 20 sorted, and so on. *The plot shows that HoneySort outperformed both Insertion sort and Quicksort for heterogeneous arrays.* Finally, Figure 7 shows that making the overlay random does not affect running time.

[8]HoneyAdapt and HoneySort are *not* equivalent; the latter is a practical adaptation of the former.



(a)



(b)

Figure 5. HoneySort: (a) On Random input. With chunk size approx. 1000 and with 6 client machines; (b) On Pre-sorted input. With chunk size approx. 1000 and with 6 client machines.

6. Summary

A new class of p2p algorithms (protocols) called "sequence protocols" can be derived systematically from multi-variable sequence equations, that are perhaps originally based on natural phenomena. Sequence protocols are self-adaptive, scalable, and fault-tolerant. They can be used to detect global triggers (Multiplicative Protocol) or enable clients in a large-scale Grid application to adapt at run-time to execute large data-sets more efficiently. HoneySort, an adaptive parallel sorting protocol, outperformed well-known algorithms Insertion Sort and Quicksort.

References

[1] D. Angluin, J. Aspnes, et al, "Computation in networks of passively mobile finite-state sensors", *Proc. ACM PODC*, 2004, pp. 290-299.

[2] R. Arpaci-Dusseau, A. Arpaci-Dusseau, D. E. Culler et al, "Searching for the sorting record: experience with NOW-Sort", *Proc. SIGMETRICS Symp. Par. and Distbd. Tools*, 1998.

[3] O. Babaoglu et al, "Design patterns from biology for distributed computing", *Proc. Eur. Conf. Complex Sys.*, Nov. 2005.

[4] Arvind, "Bluespec: A language for hardware design, simulation, synthesis and verification", *Proc. MEMOCODE*, page 249, 2003.

[5] G. di Caro, M. Dorigo, "AntNet: Distributed Stigmergic Control for Communications Networks", *Journ. AI Res.*, 9, 1998, 317-365.

[6] R. Bhagwan et al, "Total Recall: System Support for Automated Availability Management", *Proc. Usenix NSDI*, 2004.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C, Stein, *Introduction to Algorithms*, MIT Press, 2nd Ed., 2001.

[8] D. Kempe, A. Dobra, J Gehrke, "Gossip-based computation of aggregate information", *Proc. IEEE FOCS*, 2003, pp. 482-491.

[9] S. D'Silva, "Collective decision making in honeybees", Unpublished, http://www.msu.edu/user/dsilva/beepp2.ps

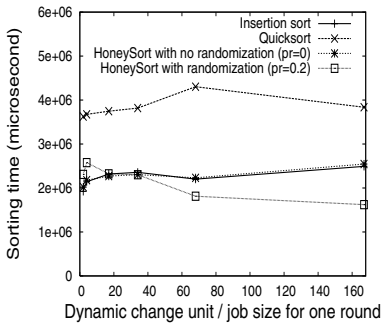


Figure 6. HoneySort on heterogeneous arrays. Input array has 1 million entries, chunk size approx. 3000 elements, 333 chunks total. With 6 client machines. Randomization prob. $pr = 0.2$. Groups of dynamic change unit (x-axis) chunks are alternately random and sorted.

Node Degree in Overlay	HoneySortRunning Time (ms)
2	945183
15	1086741
29	882870

Figure 7. Effect of Client Overlay topology on HoneySort. With 30 clients. Input array has 1 million entries, chunk size approx. 5000 elements, thus a total of 200 chunks.

[10] J. Gray, Sorting Benchmark, http://research.microsoft.com/barc/SortBenchmark.

[11] I. Gupta, "On the design of distributed protocols from differential equations", *Proc. ACM PODC*, 2004, pp. 216-225.

[12] I. Gupta and Y. Jo, "On the Use of Sequences, Phase Changes, and HoneyBees For Designing Adaptive Distributed Systems", Technical Report UIUCDCS-R-2005-2523, UIUC, Feb. 2005.

[13] M. Herbster and M. K. Warmuth, "Tracking the Best Expert", *Proc. ICML*, 1995, pp. 286-294.

[14] M. Jelasity, R. Guerraoui, A.-M. Kermarrec and M. van Steen, "The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations", *Proc. Middleware*, 2004.

[15] T.G. Kurtz, *Approximation of population processes*, Reg. Conf. Series in Appl. Math., SIAM, 1981, ISBN 0-89871-169-X.

[16] B. T. Loo et al, "Implementing Declarative Overlays " *Proc. SOSP*, 2005, pp. 75-90.

[17] A. Medina et al, "BRITE: An Approach to Universal Topology Generation", *MASCOTS*, 2001, page 346.

[18] M. Merritt, G. Taubenfeld, "Computing with infinitely many processes", *Proc. DISC*, Springer LNCS 1914, 2000, 164-178.

[19] M. Mitzenmacher, "The power of two choices in randomized load balancing", *IEEE TPDS*, 12:10, Oct. 2001, 1094-1104.

[20] M. O. Rabin, "Randomized Byzantine generals", *Proc. FOCS*, 1983, pp. 403-409.

[21] T. Seeley, *The Wisdom of the Hive*, Harvard Univ. Press, Hardcover Ed., 1996.

[22] S. H. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*, Perseus Book Group, First Ed., 2001.

[23] A. Uresin and M. Dubois, "Parallel asynchronous algorithms for discrete data", *JACM*, 37:3, July 1990, 588 - 606.

[24] S. Voulgaris, D. Gavidia and M. van Steen, "CYCLON: Inexpensive membership management for unstructured P2P overlays", *JNSM*, 13:2, June 2005, 197-217.

[25] Closed forms for the logistic map, http://www.mathpages.com/home/kmath188.htm.

[26] Mute: Simple Anonymous File Sharing, http://mute-net.sourceforge.net.