A New Class of Nature-Inspired Algorithms for Self-Adaptive Peer-to-Peer Computing

STEVEN Y. KO and INDRANIL GUPTA
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL 61801
{sko,indy}@cs.uiuc.edu

and
YOOKYUNG JO
Department of Computer Science
Cornell University
Ithaca, NY 14853
ykjo@cs.cornell.edu

We present, and evaluate benefits of, a design methodology for translating natural phenomena represented as mathematical models, into novel, self-adaptive, peer-to-peer (p2p) distributed computing algorithms ("protocols"). Concretely, our first contribution is a set of techniques to translate discrete "sequence equations" (also known as difference equations) 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. Individual processes do not simulate the source sequence equations completely. Instead, each process executes probabilistic local and message passing actions, so that the emergent round-to-round behavior of the sequence protocol in a p2p system can be probabilistically predicted by the source sequence equations. The paper's second contribution is the design and evaluation of a set of sequence protocols for detection of two global triggers in a distributed system: threshold detection and interval detection. This paper's third 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. Finally, we design a variant of HoneyAdapt, called "HoneySort", for application to Grid parallelized sorting settings using the master-worker paradigm. Our evaluation of the above contributions consists of mathematical analysis, large-scale trace-based simulation results, and experimental results from a HoneySort deployment.

This work was supported in part by NSF CAREER grant CNS-0448246 and in part by NSF ITR Grant CMS-0427089.
This paper is an extended version of our previous conference paper in IEEE SASO 2007 [Ko et al. 2007].

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.1.0 [**Models and Principles**]: General; G.3 [**Probability and Statistics**]: Probabilistic algorithms

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Complex Adaptive Systems, Bio-Inspired Techniques, Autonomic Computing and Communication, Design Methodology, Sequence Equations, Difference Equations, Distributed Protocols, Grid Computing, Sequence Protocols, Probabilistic Protocols, Adaptivity, Convergence

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. However, given a natural phenomenon, deriving a distributed protocol from it is a non-trivial task, e.g., [Babaoglu et al. 2005; Babaoglu et al. 2002; Bonabeau et al. 1999; di Caro and Dorigo 1998; Maniezzo et al. 2004; Rohrer 2007]. As a result, existing approaches for this translation (from natural phenomenon to protocol) often tend to be informal and ad-hoc, and thus the relation between phenomenon and protocol is often not quantifiable.

This paper follows an alternative approach, that is both mathematical and practical, for translating natural phenomena into novel p2p protocols. We leverage the fact that many 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 [Agarwal 2000; Seeley 1996; Strogatz 2001]. Our approach is then to create *design methodologies* [Gupta et al. 2007] that translate these mathematical models into p2p protocols whose behavior can be probabilistically predicted by the original model. We then use the generated protocols to solve practical distributed computing problems.

Abstractly, a design methodology is targeted at a particular class of mathematical models, e.g., discrete sequence equations in a particular format and structure. The methodology 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, it generates an output that is a p2p protocol, in the sense that it is an algorithm run in a decentralized manner by each process in the p2p system. The p2p protocols in this paper run in synchronized *rounds* across the entire system (we will elaborate on this soon).

Most interestingly however, some of the p2p system-wide behavior of this generated protocol can be probabilistically predicted by the set of equations input to the methodology. In other words, the following two facts are true: (1) the round-to-round expected (i.e., average case) behavior of the protocol on a distributed system-wide basis, can be predicted using the sequence equation, and (2) the round-to-round error between the actual distributed system state and the prediction from sequence equation state, decreases quickly as the total number of processes is increased. The protocol running at each process does *not* simulate the entire sequence equations set. Instead, the protocol involves simple local and message-passing actions at each process. Based on the above, we hypothesize that the trajectories

of the protocol behavior may also be predictable by the sequence equations - our simulations of specific protocols provide empirical evidence for this hypothesis.

The concrete contributions of this paper are three-fold:

I. *Design Methodologies to translate Sequence Equations into Sequence Protocols:* We describe a novel design methodology for translating sequence equations (also known as difference equations [Agarwal 2000; Strogatz 2001]) into a class of novel p2p protocols that we call *sequence protocols*. The techniques in this methodology ensure that the round-to-round emergent system-wide behavior of a sequence protocol can be probabilistically predicted by the source set of sequence equations from which it is translated. Sequence protocols are completely decentralized, and involve low and scalable per-process communication overheads and memory.

Since we show only the round-to-round prediction, we make the following hypothesis and use simulations of specific sequence protocols to provide evidence of it: the sequence equation can be used to understand the trajectories of the sequence protocol behavior. Validating this hypothesis is important because it means that sequence protocols are self-adaptive; their emergent behavior in a p2p system inherits the equilibrium points from the sequence equations. Furthermore, this is achieved without any external input, and by using only the internal protocol actions at each process.

II. *Protocols for Detecting Certain Global Triggers in a Distributed System:* In order to demonstrate the utility, properties and power of our methodology and nature-based sequence protocols, we design and evaluate two protocols for detecting certain global triggers in a distributed fashion. Specifically, our Multiplicative protocol detects when the global average of a variable crosses a threshold, while our Logistic protocol detects when the global average falls outside an interval. These two protocols are based on the classical multiplicative and logistic equations respectively. We analyze these protocols mathematically. We then experimentally study their behavior via simulation to provide evidence for our hypothesis about the trajectories and equilibrium points being the same as the sequence equations.

III. *Self-Adaptive Grid Computing Inspired by Honeybee Behavior:* Finally, we design and evaluate a new self-adaptive Grid computing protocol called *HoneyAdapt*. HoneyAdapt is a sequence protocol that is derived from sequence equations modeling adaptive bee foraging behavior in nature. HoneyAdapt is intended for master-worker style 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. We implement a variant of HoneyAdapt for the parallel sorting problem, and our system is called HoneySort. Our experimental evaluation shows that HoneySort outperforms parallel versions of classical algorithms such as Quicksort and Insertion sort (Section 5.2). This is because HoneySort is able to adapt itself to changing input data characteristics due to its sequence equation-based behavior, while the classical algorithms treat all data homogeneously. Finally, the Multiplicative protocol and the Logistic protocol can be used to detect partitioning of a random overlay, or to monitor the resource consumption system-wide (Section 3), thus enabling distributed applications to be reactive to such changes.

An advantage of our approach in this paper is that we are translating a mathematical model into a p2p protocol, without using any other innate characteristics of the natural phenomenon. This provides us systematic translation techniques, so that the properties of the derived p2p protocol can be analyzed. Further, it is generic enough to allow us to translate mathematical model equations that may not even be derived from natural phenomena, into the corresponding p2p protocols.

Other Related Work: The class of sequence protocols we discuss are related to Markov chains, yet our work is different because it deals with algorithm design. The work in [Uresin and Dubois 1990] has each process simulate the entire sequence equation, while our sequence protocols create an emergent state that can be probabilistically predicted by the sequence equations.

Previously in [Gupta et al. 2007], 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, e.g., see [Agarwal 2000; Strogatz 2001].

Population protocols [Angluin et al. 2006; Merritt and Taubenfeld 2000] 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 [Kurtz 1981]. Methodologies in general have been used to systematize the design process in many fields, e.g., [Arvind 2003], but in distributed computing, they have begun to emerge only recently, e.g., [Loo et al. 2005].

Like our basic sequence protocols, many classical distributed algorithms also operate in rounds, e.g., [Rabin 1983]. Other classical algorithms that define state machines at each process have been the focus of the distributed computing community for many decades, e.g., [Misra and Chandy 1982; Chandy and Lamport 1985]. The reader is encouraged to see the excellent tutorial on state machines in [Schneider 1986]. Many of these algorithms operate asynchronously. Distributed protocols have been specified using declarative languages, e.g., the recent P2 system [Loo et al. 2005] generates p2p overlays declaratively. Mitzenmacher used differential equations to analyze distributed protocols for load balancing [Mitzenmacher 2001]. However, to our knowledge, none of this work looks at generating p2p protocols directly from mathematical models.

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 HoneySort 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 [Voulgaris et al. 2005] or via a peer-sampling service [Jelasity et al. 2007].

Section 2 describes the new design methodology. A case study of two simple sequence protocols for detecting global properties 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

In this section, we present an innovative design methodology that translates certain types of sequence equations into new p2p protocols which we call *sequence protocols*. While this section is focused on describing only the generic methodology, Section 3 will then provide examples of two simple protocols derived from this methodology.

The canonical single-variable sequence equation can be expressed in the form:

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

Here, k is a constant non-negative integer (representing memory of the equation), m takes on integer values $\geq k$, and all x_i's take real values $\in [0, 1]$. f is a deterministic function with a finite number of terms, and a range that lies within $[0, 1]$. Henceforth in the paper, when the output of f is < 0, we will take it to be 0, while if the output is > 1, we will take it to be 1. The types of terms allowed in f will be enumerated in Section 2.2.

For clarity of notation in sequence equations, we will use a subscripted index (e.g., x_i) to represent the time index i for a given variable. Time, in these equations, thus takes discrete positive integer values.

We also extend our methodology to multi-variable sequence equations with a finite number of variables (say, l variables). We use a side-index (e.g., xj) to denote the j^{th} variable in a multi-variable equation [1]. 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. Thus, xj_i denotes the value of variable xj at time i. Hence, the canonical multi-variable sequence equation

[1]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.

would have for each variable xj an equation defined in the form:

$$xj_{m+1} = fj(x1_m, x1_{m-1}, \ldots, x1_{m-k}, x2_m, \ldots, x2_{m-k}, \ldots, xl_m, \ldots, xl_{m-k})$$

The sequence protocol derived from a system of sequence equations is an algorithm that runs in a decentralized manner at each process in the p2p system. Like the original sequence equations, a sequence protocol operates in system-wide *rounds*, where each process is synchronized to 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*, where each state variable takes binary values. This tuple is denoted as $< s1(p), s2(p), \ldots, sl(p) >$, where the value of the j^{th} bit $sj(p)$ represents whether the process is in state sj (if state variable $sj(p)$ is 1) or not.

Finally, we use the term sj_i to denote the *system-wide fraction* of processes in state sj at the start of round i (for $1 \leq j \leq l, i \in Z^+$). To summarize, $sj_i(p)$ is a binary variable (denoted as "state variable sj at process p" at time i), while sj_i is a system-wide real number $\in [0, 1]$. Intuitively, each sj_i in the sequence protocol corresponds to xj_i, as elaborated below.

The relation between the sequence equations and the sequence protocols is defined by the following two conditions in any round m: (1) for any j, the sequence equations can be used to predict the *expected* round to round behavior of the sequence protocol. In other words, suppose we replace each xj_i $(1 \leq j \leq l)$ on the right hand side of the sequence equation (i.e., in $fj(.)$) with the corresponding actual value of sj_i. Then, the derived value of xj_{m+1} on the left hand side will predict the expected value of sj_{m+1}; and (2) as N is increased, the error between predicted xj_{m+1} and actual sj_{m+1} above, varies as $o(1)$.

Once these conditions are shown, they allow us to make the hypothesis that the sequence protocol may have the same time-based trajectories (i.e., behavior over multiple rounds) and equilibrium points as the source sequence equations. Notice that this means that processes will run local and message-passing actions that make the *emergent* (i.e., system-wide) behavior in the distributed system the same as the sequence equations. We will show the above conditions (round-to-round prediction) mathematically in the following discussion, and provide evidence for the hypothesis (trajectory prediction) via simulations of specific protocols in the following sections.

The above notations and correspondences are summarized in Table I for the reader's convenient reference.

2.1 Basic Translation Methodology

For simplicity of exposition, we focus our discussion on single-variable sequence equations only. We will comment at appropriate places on the generalization to multi-variable equations. Consider the single-variable equation:

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

Here, k is a constant non-negative integer (representing memory of the equation), all $x_i \in [0, 1]$, and f has a range that lies within $[0, 1]$. m is a notation standing for

Table I. Correspondences between terminologies of sequence equation and the derived sequence protocol. The last two columns are explained in Section 2.1.

Sequence Equation	Interpretation	Sequence Protocol	Interpretation
x or xj	Denotes a variable	s or sj	Denotes a state variable
-	-	$sj_i(p)$	Binary variable - whether process p is in state sj at ith round start
xj_i	Value of var. xj at ith round start	sj_i	Fraction of processes in state sj at ith round start
xj	Current value of variable xj	sj	Fraction of processes currently in state sj
$x_{m+1} = f(x_m, x_{m-1}, \ldots, x_{m-k})$	Sequence equation (single var.)	$s_{m+1} = f(s_m, s_{m-1}, \ldots, s_{m-k})$	Predicted sequence protocol behavior (single var.)
Variable	-	State variable	-
Term	-	Token Generation (then relay+apply)	-

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), where each term is deterministic and involves a subset of the variables $x_m, x_{m-1}, \ldots, x_{m-k}$. Section 2.2 shows that terms are allowed to be multi-variable, and either polynomial or non-polynomial. Given this, our design methodology converts: (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 a boolean value, indicating whether the process is in that state or not. For process p, we call this as the "state variable s at p". When the state variable at process p is 1, we say that "p is in state s"; when the state variable is 0, we say that "p is out of state s".

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 the sequence equations predict the expected fraction of processes in state s. Concretely, suppose it is true that: (i) the *fraction* of processes in state s, at the start of a given round, is the value s_m, and (ii) that fraction of processes in state s, at the start of the immediately previous $k-1$ rounds were respectively $s_{m-1}, \ldots, s_{m-k+1}$. Then, the protocol actions must ensure that the expected fraction of processes in state s at the start of the following round will be predicted by:

$$E[s_{m+1}] = f(s_m, s_{m-1}, \ldots, s_{m-k})$$

Overview of the Generated Sequence Protocol: The generic framework of a sequence protocol, involving only one variable x, is illustrated in Figure 1 for completeness. We elaborate below. First, each process p maintains the following state variables at any time: (i) $s(p)$, the state of process p in the current round, (ii) $s_{-1}(p), s_{-2}(p), \ldots, s_{-k}(p)$, the states of p in the immediately previous k rounds, and (iii) $s_{next}(p)$, a running variable for the state of p for the next round. $s_{next}(p)$ is continuously updated during the current round (based on actions and messages described below). For multi-variable sequence equations, (i) – (iii) are maintained for each of the variables.

Second, at the start of each round at process p, the following initializing actions

```
At process p:
    boolean s(p); // state at start of this round
    boolean s_{-1}(p), ..., s_{-k}(p); // states at start of last k rounds
    boolean s_{next}(p); // next state - running variable
During a Round at process p:
  int numtokens; //number of tokens
    for (i = k down to 2)
        set s_{-i}(p) := s_{-(i-1)}(p); // remember past states
    set s_{-1}(p) := s(p); // remember the last state
    set s(p) := s_{next};
    set s_{next}(p) := 0;
    Use Token Generation Algorithm(f, s(p), ..., s_{-k}(p)) to generate tokens;
    let numtokens be the net local tokens after subtracting negative tokens from positive tokens;
    if (numtokens ≠ 0)
        select |numtokens| random and distinct non-faulty processes;
            if (numtokens > 0)
                send one positive token to each chosen process;
            else
                send one negative token to each chosen process;
  //Token Relay and Apply
    while (round is not over)
        whenever received a token
            if (received token is positive)
                if (s_{next}(p)==1)
                    send one positive token to a random non-faulty process;
                else
                    set s_{next}(p) := 1; // consume token
            else// if(received token is negative)
                if (s_{next}(p)==0)
                    send one negative token to a random non-faulty process;
                else //if(s_{next}(p)==1)
                    set s_{next}(p) := 0; // consume token
```

Fig. 1. **Sequence Protocol: A Generic Framework.** This sequence protocol is derived from a single-variable sequence equation with constant or polynomial terms. The corresponding `Token Generation Algorithm` is described in Section 2.2.1. An extension of this protocol can be used for translating sequence equations with non-polynomial terms (Section 2.2.2).

are taken: (i) for each $i = k$ to 2, $s_{-i}(p)$ is replaced by the value of $s_{-(i-1)}(p)$, (ii) $s_{-1}(p)$ is set to $s(p)$, (iii) $s(p)$ is set to $s_{next}(p)$, and (iv) $s_{next}(p)$ 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 previous 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 to zero 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[2].

In each round, process p executes two types of actions – *Token Generation*, followed by *Token Relay and Apply*. Token generation occurs independently at each process. It creates a number of *token* messages, based on the terms in the sequence equations. Token messages are then *relayed* to other processes through

[2]Although we assumed perfect synchronization, in real life, small synchronization errors may cause messages from older rounds to be received - these messages would need to be dropped.

random walks. In turn, when p receives tokens, it *applies* these tokens to $s_{next}(p)$. 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 each of the the N processes create per round, for each positive (resp. negative) term T, an expected number of T positive (resp. negative) tokens. This latter value of T is calculated by substituting each x_i (or xj_i) with the corresponding s_i (or sj_i), i.e., the fraction of processes in state s at time i.

(2) **Token Relay and Apply:** After generating tokens, a process forwards each token to a random non-faulty target process. When process p, with $s_{next}(p) = 0$, receives a positive token, it *consumes* the token by setting $s_{next}(p)$ to 1. When a process with $s_{next}(p) = 1$ receives a negative token, it consumes the token by setting $s_{next}(p)$ to 0. If none of these two conditions apply, the process forwards the received token to another random non-faulty target process. Thus, each token takes a random walk until it is consumed.

The above operations - token generation, relay and apply - generalize in a straightforward manner for a multi-variable sequence equation in variables $x1, x2, \ldots, xl$. Each process executes, in each round, the above operations independently and separately for each of the l state variables $s1(p) \ldots, sl(p)$. In our analysis of the methodology below, for generality, we will use the multi-variable equation terminology.

Relation Between Distributed system state and Sequence equation state: Before describing the token generation actions for individual mathematical terms, we show the relation between the round-to-round system-wide behavior of the sequence protocol and the sequence equation's behavior. This discussion assumes certain preconditions. Section 2.2 and the token generation, relay, and apply operations (see above) satisfy these preconditions.

Consider a multi-variable sequence equation system in l variables $\{x1, x2, \ldots, xl\}$ ($xi \in [0, 1]$), and a sequence protocol so that: (A) all processes execute the same probabilistic algorithm independently, thereby producing tokens whose number is independent and identically distributed across processes; (B) for each term T appearing in the sequence equation, the expected number of tokens generated, per round and per process, can be derived as the value of term T itself when each xj_i in it is replaced with the corresponding sj_i's value; all such tokens are positive if T is positive and negative otherwise; (C) the tokens are relayed and applied until the system has quiesced, i.e., the number of consumed tokens cannot increase further.

We show two facts below. Firstly, we argue that from one round to the next round, under the above preconditions, the sequence equations can be used to predict the *expected* fractions of processes that will be in states $\{s1, s2, \ldots, sl\}$, at the start of the next round. In other words, suppose we are currently in round m. Now, replace each xj_i by the corresponding value of sj_i (the fraction of processes in state sj at round i's start) $\forall i, j$, on the right hand side of the sequence equation $fj(.)$. Then, it is true that the derived value of xj_{m+1} on the left hand side will be the same as the expected value of sj_{m+1}, the fraction of processes in state sj at the start of round $(m + 1)$. Notice that by itself, this argument holds only from one

round to the next round (and not beyond).

However, secondly, we also show below that the error between the values of xj_{m+1} and actual sj_{m+1} decreases as N increases. These two facts will lead us to hypothesize that the trajectories (over multiple rounds) of the distributed system state $< s1_i, s2_i, \ldots, sl_i >$ may in fact closely follow the sequence equation behavior $< x1_i, x2_i, \ldots, xl_i >$. Our simulations in the following sections will provide evidence for this hypothesis.

To show this, we first consider what happens at each process. The following discussion applies to each variable index $j \in \{1, 2, \ldots, l\}$. In the current round m (i.e., at the start of the round), the fraction of processes in state sj is sj_m. In the immediately preceding k rounds, the fractions of processes in state sj were $sj_{m-1}, \ldots, sj_{m-k}$ respectively. Then, during the current m^{th} round, adding all positive and negative tokens for state sj, and using the precondition (B) from above, it turns out that the *expected* number of tokens generated at any one given process, for sj, is $= fj(\{sh_i\}_{i=m,h=l}^{i=m-k,h=1})$.

Next, to analyze what happens with tokens generated throughout the N processes in the system, we use the Central Limit theorem. Consider d independent and identically distributed random variables $Y_1, Y_2, \ldots, Y_d$, each with finite values of mean μ and standard deviation σ. The Central Limit Theorem states that as d increases to infinity, the distribution of the quantity $(\frac{Y_1+Y_2+\ldots+Y_d}{d})$ approaches the normal distribution with a mean μ and standard deviation $\frac{\sigma}{\sqrt{d}}$, irrespective of the shape of the original distribution.

In our case, we set $d = N$, and $Y_g =$ the number of tokens generated at the g^{th} process p_g. The Y_i's are all independent random variables with the same distribution (due to precondition (A) above). The average of each Y_i is merely $fj(\{sh_i\}_{i=m,h=l}^{i=m-k,h=1})$. Finally, when these tokens are relayed and applied until the number of tokens consumed cannot increase further (precondition (C) above), the *probability* of a process having $sj_{next}(p) = 1$ at the end of the m^{th} round, is the quantity $E[\frac{Y_1+Y_2+\ldots+Y_d}{d}]$, which also is $fj(\{sh_i\}_{i=m,h=l}^{i=m-k,h=1})$.

To wrap up, we briefly revisit the extreme cases when $fj(\{sh_i\}_{i=m,h=l}^{i=m-k,h=1}) \notin [0, 1]$. If this quantity is < 0, then on expectation only negative tokens are left at the end of the round, while if this quantity is > 1, then on expectation more than N tokens are generated and due to quiescence every process gets at least one token. When this round is over, the former thus ensures that *no* processes are in state sj, while the latter ensures that *all* processes are in state sj.

Putting the discussion so far together, we conclude that the expected round-to-round behavior of the distributed system state is the same as the behavior of the sequence equation. However, from one round to the next round, the *actual* value of sj_i may be different from the calculated xj_i – this error arises solely from the standard deviation of the distribution of $(\frac{Y_1+Y_2+\ldots+Y_d}{d})$. For instance, if the standard deviation were zero, then preconditions (A), (B), and (C) would imply that sj_i always exactly matched xj_i in each round i. On the other hand, a larger standard deviation means that it is more likely that too few or too many tokens are generated, and thus the actual sj_i is farther from the expected xj_i.

Hence we use the standard deviation of $(\frac{Y_1+Y_2+\ldots+Y_d}{d})$ as representative of the error between the distributed system state sj_i and the sequence equation state xj_i.

Since the standard deviation is $O(\frac{1}{\sqrt{N}})$, this means that as N becomes larger and larger, the sequence protocol behavior matches the sequence equation more and more closely. Thus the trajectories (over multiple rounds) of the sequence equation and the distributed system state are likely to resemble each other more and more as N increases towards infinity.

A few words are due on inheritance of equilibrium points in the original sequence equation: an unstable equilibrium point in the original equation behavior will lead to a sequence protocol that always diverges away from the equilibrium point. This is desirable, and occurs because even the smallest perturbation (arising from the error) can cause the system to diverge away. However, we hypothesize that a stable equilibrium will be inherited as sequence protocol behavior that always stays converged around the stable equilibrium point - this is also desirable. Proving these hypotheses is beyond our scope here, however the following sections use simulations of actual sequence protocols to show evidence that these hypotheses may be true.

Quiescence Time for Dissemination of Tokens: We analyze below the quiescence time, i.e., time to satisfy precondition (C) above. Quiescence time for a round is defined as the time to disperse all tokens, generated system-wide at the start of the round, until the number of tokens consumed cannot increase anymore in this round. We show that the quiescence time is $\theta(\log(N))$ with high probability (w.h.p.). This latency time determines the length of the token relay and apply phase, and thus the duration of a round in the sequence protocol.

Recall each process generates tokens independently and with identical distributions. Thus, let us assume that the number of tokens generated system-wide is $(f \cdot N)$, with $f \geq 0$. Now, notice that if all these tokens are the same sign (either all positive or all negative), then the quiescence time will be longer than if some tokens are negative and others positive. This is because positive and negative tokens cancel each other out (see Token Relay and Apply in Figure 1), and thus this situation would quiesce faster. Hence we analyze the worst-case by assuming, without loss of generality, that all these $(f \cdot N)$ tokens are positive. In calculating the quiescence time below, we assume that time is normalized in terms of the average packet latency in the network.

The value of f leads us to two different cases:

1. $f > 1$: Since there are more tokens than processes, the system quiesces when each process has consumed at least one token. In order to analyze the time for this to happen, consider a process p' which has received no tokens yet (and thus has consumed none). Since at most N of the $f \cdot N$ tokens could have been consumed (there are only N processes), there are at least $(f-1) \cdot N$ tokens that are being forwarded randomly (according to the token relay rules above). The probability that p' will *not* receive any of these $(f-1) \cdot N$ tokens within the next time unit is $\leq (1 - \frac{1}{N})^{(f-1)\cdot N} \simeq e^{-(f-1)}$.

Next, we calculate what happens after a latency of $\frac{f}{f-1} \cdot log(N)$ time units. The probability that p' will *not* receive any token within $(\frac{f}{f-1} \cdot log(N))$ time units is $\leq e^{-(f-1)\cdot\frac{f}{f-1}\cdot log(N)} = e^{-f\cdot log(N)} = N^{-f}$. Thus, the probability that *all* the processes in the system *will receive at least one token* within $(\frac{f}{f-1} \cdot log(N))$ time units is simply $\geq (1 - N^{-f})^N \simeq 1 - \frac{1}{N^{f-1}} = 1 - o(1)$, since $f > 1$. Thus, w.h.p., it

takes $\frac{f}{f-1} \cdot log(N)$ message transmission times for all processes to consume a token, and hence for the system to quiesce.
2. $0 \leq f < 1$: This means that the system quiesces when exactly $f \cdot N$ processes have received one token each (and have consumed them), with the remaining processes having received no tokens at all. We calculate the quiescence time by analyzing the time for all of these $f \cdot N$ tokens to be consumed. Consider an unconsumed token t' that will next be forwarded to a random process. Since at most $f \cdot N$ processes at any time could have consumed a token (there are only $f \cdot N$ tokens in total), with probability $\geq (1 - f)$, token t' will be forwarded to a process that has consumed no token. Thus, every time a token is forwarded, it has a probability $\leq f$ of *not* being consumed.

Next, we calculate what happens after a latency of $(2 \cdot log_{\frac{1}{f}}(N))$ time units. The probability that the token t' will not be consumed by this time is $\leq f^{2 \cdot log_{\frac{1}{f}}(N)} = \frac{1}{N^2}$. Thus, the probability that *all* $(f \cdot N)$ tokens *will* be consumed by this time is:

$$\geq (1 - \frac{1}{N^2})^{f \cdot N} \simeq e^{-\frac{f}{N}} \simeq 1 - \frac{f}{N} \geq 1 - \frac{1}{N}$$

Here, we have used the fact that N is large, and that $f < 1 \ll N$. Therefore, we have shown that w.h.p., it takes $(2 \cdot log_{\frac{1}{f}}(N))$ time units for all tokens to be consumed, and for the system to quiesce.

Henceforth, our mathematical analysis of sequence protocols 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. However, notice that in the above calculation, we ignored the case of $f = 1$; this is reasonable since the probabilistic token generation rules make it very unlikely that exactly N tokens will be generated. In addition, although $f = 1$ means that the very last token would require an expected $O(N)$ rounds to be consumed, one can use a shorter round duration to achieve a distributed system state that is arbitrarily close to the quiesced state. Specifically, with the round duration chosen as $(2 \cdot log_{\frac{1}{q}}(N))$ time units (where $q \in (0, 1)$), using the same logic as in case (2) above, $(q \cdot N)$ processes would have received at least one token w.h.p. by the round's end. Thus, by setting q as close to 1.0 as desired, one can bring the sequence protocol arbitrarily close to the sequence equation behavior. Our experiments in Sections 3 and 5 show that fixed round lengths do not affect real-life convergence properties of sequence protocols.

Finally, notice that although processes are synchronized at the start of each round, no synchrony is required within the round itself. This is because once tokens are generated at the start of the round (this requires synchrony), the token relay and apply phase can be executed asynchronously at each process.

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 the discussion in the previous section to hold, we require preconditions (A) and (B) of Section 2.1 to hold, i.e., for each term T in the sequence equations, each process should use identical actions that sample the same probability distributions, and independently

generate an expected T tokens at the start of the round. We describe below how polynomial, multi-variable, and non-polynomial terms, all satisfy this requirement.

2.2.1 *Polynomial Terms.*

2.2.1.1 *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. This ensures that each process creates an expected r tokens. Notice that this token generation action requires no message exchange, since sampling actions are completely local.

2.2.1.2 *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 s. If indeed $s_{-j}(p) = 1$, then p generates an average of r tokens, in a similar manner to the constant term translation in the last paragraph. If $s_{-j}(p) = 0$, then p generates no tokens. If the current round is m, then the probability that a process satisfies $s_{-j}(p) = 1$ is s_{m-j}, thus the expected number of tokens generated at any process is $s_{m-j} \cdot r = T$. This token generation action requires no message exchange either.

Ex. 1: The multiplicative equation $x_{m+1} = r \cdot x_m$ can be translated using this token generation rule, and is discussed in Section 3.1.

2.2.1.3 *Multiplicative terms of form $T = r \cdot \Pi_{i=m}^{i=m-k} x_i^{(j_i)}$ (all j_i's are non-negative, with at least one j_i being strictly 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 s, i.e., if $s_{-(m-i')}(p) = 1$ (where $s_0(p) = s(p)$). 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 during the last k rounds.

Once all acknowledgements are received at process p, it 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 $T/(r \cdot s_{i'}) = s_{i'}^{(j_{i'}-1)} \cdot \Pi_{i=i'-1}^{i=m-k} s_i^{(j_i)}$, 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 $T/(r \cdot s_{i'})$, and the fraction of processes with $s_{-(m-i')}(p) = 1$ is $s_{i'}$, the expected number of tokens generated at the process is $r \cdot s_{i'} \cdot T/(r \cdot s_{i'}) = T$.

Ex. 1: If $T = 6 \cdot x_m^2 \cdot x_{m-3}^2$, then $i' = m$, $r = 6$, and $T/(r \cdot x_{i'})$ is written as $x_m \cdot x_{m-3} \cdot x_{m-3}$. Each process p in state s at the round start sends out 3 sampling messages. If the first received reply is from a process in state s, and the remaining two replies are from processes that were in state s at the start of 3 rounds ago, then an average of 6 tokens are generated by p for this round.

Ex. 2: The logistic equation $x_{m+1} = r \cdot x_m \cdot (1 - x_m)$ can be translated using this token generation rule, and is discussed in Section 3.2.

2.2.1.4 *Translating Sequence Equations with Multiple Variables.* The presented methodology (with techniques from both Sections 2.2.1 and 2.2.2) can be generalized to translate a multi-variable system of sequence equations. We briefly summarize here. Consider a multi-variable sequence equation in $l > 1$ variables $x1, x2, \ldots, xl$. In the derived sequence protocol, for each variable xj $(1 \leq j \leq l)$, each process maintains one state variable sj, along with a memory of the last k values of it. Thus, the process state is a $k-$tuple of $l-$ tuples. During each round, the process executes separate token generation actions for each fj, independent of one other, i.e., actions for each sj are independent of each other. As the reader can expect, the actions for sj can be derived by applying the other techniques within this Section 2.2 to the terms within fj. If fj also contains terms with foreign variables xh_* $(h \neq j)$, then one can generalize the token generation actions for sj in a straightforward manner. Finally, we ensure that the tokens generated from fj are tagged with j so that they are applied only to state variables sj.

2.2.2 *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 within each subround. The subrounds occur consecutively one after another within the round, do not overlap with each other (with an exception explained below), and have fixed durations. Processes are assumed to be synchronized at the start of each subround. Subtokens do not affect state variables directly, but instead contribute towards eventual generation of tokens - we elaborate in the following paragraphs how this works for individual terms. Given multiple terms in a sequence equations f, each may require a different number of subrounds - Section 2.2.2.4 explains that these subrounds are tied together so that the last subround of all terms overlap with each other (this is the exception). Ignoring such tying issues for now, we focus only on individual terms below.

2.2.2.1 *Division terms $T = r/x_{m-i}$ $(0 \leq i \leq k$, with $x_0 \equiv x)$.* In this description, we use a shorthand notation of $s_0(p')$ for $s(p')$, allowing x_0 to appear inside the term. To translate such a term, split each round into two subrounds. In the first subround, process p, regardless of its state at the round start, iteratively selects a target process uniformly at random. This selected process, say p', acknowledges with the value of its local $s_{-i}(p')$. p continues iterating as long as the target process p' continues to reply with $s_{-i}(p') = 0$. Process p stops when it receives the first reply that satisfies $s_{-i}(p') = 1$. Process p keeps the count of the number of acknowledging targets. If their total is z, p generates an average $r \cdot z$ tokens. In the second (final) subround, these tokens are relayed and applied as usual.

Since the probability of a random target process p' being in state $s_{m-i}(p')$ is s_{m-i}, the expected number of tries until this event happens is $E[z] = 1/s_{m-i}$. Thus, the expected number of tokens generated at any process is $r \cdot 1/s_{m-i}$, as desired.

Finally, we end with a word on the restrictions for specifying this term. Since $x_* \in [0, 1]$, the division term M has to be structured so that either: (a) it has a value lying in a subinterval of $[0, 1]$ (e.g., the equation $x_{m+1} = \frac{1/2}{x_m}$ ensures that if $x_m \in [\frac{1}{2}, 1]$, then it is also true that $x_{m+1} \in [\frac{1}{2}, 1]$), or (b) the sequence equation in which T appears has other additive terms that always ensure the right hand side

lies within some subinterval of $[0, 1]$.

2.2.2.2 *Fractional Terms.* In a sequence equation with l distinct variables that are $x1, x2, \ldots, xl$, consider a canonical term $T = r \cdot \frac{\sum_{j=1}^{j=l} b_j \cdot a_j \cdot xj}{\sum_{j=1}^{j=l} a_j \cdot xj}$, 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, we split the round into two *subrounds*. In the first subround at process p, for each j such that $sj(p) = 1$, 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 v randomly selected other processes in the group. In the second subround, p selects one subtoken at random from those that it received. Suppose the tagged value of this token is j'. One of two cases is possible: – (i) if $b_{j'}$ (in term T) is 1, then process p generates an average r tokens; (ii) if $b_{j'}$ is 0, then p generates no tokens[3].

We analyze the behavior of this protocol. In the first subround, each process receives via multicast, for each j, an expected $(v \cdot N \cdot a_j \cdot sj)$ subtokens that are tagged with j. The probability that the condition (i) above is true (i.e., the process generates tokens at all) is then =

$$\frac{\sum_{j=1}^{j=l} b_j \cdot v \cdot N \cdot a_j \cdot sj}{\sum_{j=1}^{j=l} v \cdot N \cdot a_j \cdot sj} = \frac{\sum_{j=1}^{j=l} b_j \cdot a_j \cdot sj}{\sum_{j=1}^{j=l} a_j \cdot sj}$$

Since the process generates an average r tokens under this condition (and no tokens if (ii) is true), the expected number of tokens generated at each process is $T/r \cdot r = T$ (with each xj substituted by the corresponding sj).

The above actions can be made further scalable by normalizing a_j values, i.e., by dividing all a_j values by a common and large normalization constant. This normalization will not change the value of term T (since a_j's appear in all subterms within T). At the same time, it will reduce the number of subtokens ($\sum_{j=1}^{j=l} v \cdot N \cdot a_j \cdot sj$) received at each process. One should choose a normalization constant that is neither too small (to reduce message overhead) nor too large (to ensure that enough tokens are generated, and the expected equation-based behavior in fact occurs).

2.2.2.3 *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 tokens per round at any process. Ex: consider the term $T = \frac{x_m^2}{x_m^2 + x_{m-1}}$. Given such 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.

Such a term T can be translated 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

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

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. Ex: $T = \frac{x_m^2}{x_m^2+x_{m-1}}$ is translatable since a dummy variable $g = x_m^2$ can be used to write it as $T' = \frac{g}{g+x_{m-1}}$. Notice that each of g and T' is now translatable. The translation works in three sub-rounds: the first sub-round translates x_m^2 as a multiplicative term, finishing with an expected fraction x_m^2 of processes containing a g-subtoken. The second and third sub-rounds work just like a regular fractional term (Section 2.2.2.2), with each process generating a subtoken for term x_m^2 iff it was left with a g-subtoken at the end of the first round.

2.2.2.4 *Dealing with Multiple Terms with Subrounds.* If sequence equation f has multiple complex terms, the above discussion may lead to a protocol where multiple terms each need an action with multiple subrounds, e.g., consider $f(.) = \frac{1}{x_m} - \frac{1}{x_{m-1}}$. For the sequence equation behavior to be followed under this scenario, the subround splitting for each individual term will need to be normalized to have a *common last subround*. In this common last subround, all the tokens generated by *each* term's translation can be relayed and applied throughout the system. For instance, in the above example, each of the terms $\frac{1}{x_m}$ and $\frac{1}{x_{m-1}}$ will have a distinct first subround, but together they will share the same second subround. In this last subround positive tokens for $\frac{1}{x_m}$ and negative tokens for $\frac{1}{x_{m-1}}$ will be relayed together (and be able to cancel each other out). In the example of Section 2.2.2.3, the last overlapping subround of g and T' was itself split into two subrounds due to the fractional nature of T'.

The token generation actions described above, for each term T, are executed independently and identically at each process, and result in an expected T tokens being generated at the process. Thus, they satisfy the preconditions (A) and (B) of Section 2.1.

3. TWO CASE STUDIES: DETECTING GLOBAL TRIGGERS

In order to demonstrate the power of the design methodology we have just described, we first consider two case studies of sequence protocols that are based on simple sequence equations, but solve important problems. In a self-adaptive distributed system, we desire to detect certain *global triggers* in a decentralized fashion. Concretely, suppose each process p proposes a value R_p for a parameter of interest R. We assume that R takes values in a finite range $[0, R_{max}]$, for some $R_{max} > 0$. Global triggers are of two types:

(1) *Threshold Trigger*: This trigger detects when the system-wide average $R_{avg} = \sum_p R_p/N$ is above or below a pre-specified threshold value. 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* which $\in [0, R_{max}]$.

(2) *Interval Trigger*: This trigger detects when R_{avg} (defined above) is inside or outside a pre-specified constant interval. The pre-specified interval $\subset [0, R_{max}]$.

Our presentation and analysis considers only one-shot variants of these global trigger detectors, however our experiments will evaluate the continuous versions of these detectors. These triggers have multiple uses. For instance, a threshold trigger can be used to detect 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, $thresh = \Omega(\log(N))$). We solve this problem via the Multiplicative protocol, described in Section 3.1.

On the other hand, the interval trigger can be used to detect whether load balancing algorithms are working correctly by monitoring whether the average per-node number of files stored (e.g., in a distributed file system) is staying within specified minimum and maximum values. We solve this problem via the Logistic protocol, described in Section 3.2.

Existing work on aggregation (e.g., [Kempe et al. 2003]) can be used to detect triggers, but we are interested in a protocol that avoids full aggregation, yet has comparable convergence times to [Kempe et al. 2003], i.e., $O(\log(N))$ rounds. In addition, aggregation protocols such as [Kempe et al. 2003] are typically one-shot and have to be run periodically, while our protocols can be run in either a one-shot mode or a continuous mode (we do the latter in our experiments).

The insight to implementing these triggers comes from the fact that given sequence equations in l variables sequence $\{x1, x2, \ldots, xl\}$, Section 2.1 showed that the corresponding sequence protocol with l state variables, will have its round-to-round behavior probabilistically predicted by the equations. Our study in this section provide evidence for our hypothesis that the fraction of processes in relevant states i.e., $\{s1, s2, \ldots, sl\}$ also span similar time-based trajectories as the sequence equation, and have the same equilibrium points in the phase space $[0, 1]^l$. A fact to consider is that these multiple equilibrium points for the sequence equation (and thus the sequence protocol) vary in both number and nature (i.e., stable versus unstable versus mixed) as the constants in the sequence equations are changed. These changes are called *phase changes* or *bifurcations*. This suggests that the sequence protocol derived from such an equation system can be used to detect triggers by mapping the trigger metrics to the constants appearing in the protocol.

3.1 The Multiplicative Protocol as Threshold Trigger Detector

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 rule for polynomial linear translation (Section 2.2.1.2). Each process maintains one state variable s. The resulting sequence protocol has only one action per round per process. At the start of the round, the process checks if it is in state s - if yes it generates an average of r tokens, otherwise it generates no tokens. The tokens are relayed and applied in the usual manner as described in Section 2.

The emergent behavior of the protocol can be analyzed via the sequence equation itself. Equilibrium occurs when $s_{m+1} = s_m$, i.e., $s = r \cdot s$, or $s \cdot (1 - r) = 0$. For all $r \neq 1$, the only equilibrium point is $s = 0$. For stability of equilibria, we use a well-known result from Lorenz [Strogatz 2001]:

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

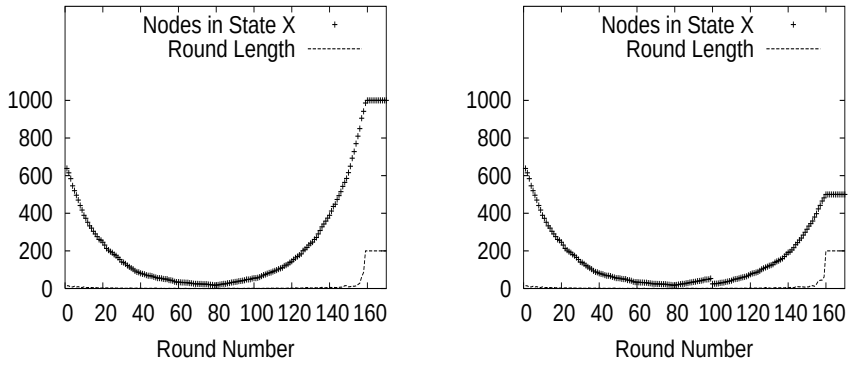


Fig. 2. **Multiplicative Protocol: Basic Phase change and Massive Failure.** (a) The value of r_{avg} changes from 0.95 to 1.05 at $t = 80$. $N = 1000$. (b) Same as (a), except that 50% of the processes fail at $t = 100$. Convergence occurs at $t = 206$ (not shown in figure). $N = 1000$.

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

Convergence Times: We analyze the number of rounds it takes for the protocol to converge. Let S denote the number of processes in state s. For $r < 1$, with $S = N$ initially, $E[S]$ after t rounds is $= (N \cdot r^t)$. The expected convergence time towards $S = 0$ is $\log_{1/r}(N)$ rounds. For $r > 1$, $S = 1$ initially implies that $E[S] = r^t$ after t rounds, giving an expected convergence time of $\log_r(N)$, towards $S = N$.

Average Threshold Trigger Detection: To implement a threshold trigger $thresh$ for metric R, we use a modified Multiplicative protocol where each process p uses a value of $r_p = \frac{R_p}{thresh}$ instead of r within the sequence protocol. Suppose $r_{avg} = \sum_p(r_p)/N$. If s_m is the fraction of processes in state s at the start of round m, since the set of processes in state s is random and has no correlation with the values of r_p's (due to the token relaying in Figure 1), we can say that the expected total number of tokens generated system-wide is $= s_m \cdot N \cdot r_{avg}$. In other words, the expected number of tokens generated per process $= s_m \cdot r_{avg}$, which produces the same global behavior as a multiplicative protocol with $r = r_{avg}$. Thus, the expected fraction of processes from one round to the next in this modified Multiplicative protocol can be probabilistically predicted by using the multiplicative equation, with r replaced by r_{avg}. Notice that the modified Multiplicative protocol is in fact an extension of, and subsumes, the pure Multiplicative protocol (the pure protocol can be derived when one sets $r_p = r$ at all processes).

Thus, when $r_{avg} < 1$, the system stabilizes with all processes out of state s; when $r_{avg} > 1$, the system converges and stabilizes with all processes in state s. Thus, detection can be achieved at an initiating process by querying, after enough time for convergence, the current states of a small *control group* of processes selected

randomly. If a majority of processes in the control group are found to be in state s, then the initiator concludes that R, the system-wide average of all R_p's, is $> thresh$, otherwise it concludes that this average is $< thresh$.

While the description of the Multiplicative protocol so far has been of the one-shot variety, we simulate the continuous version of the protocol below to show its responsiveness. This latter variant requires periodic forcing of a few processes into state s, and a few others out of state s. This is to avoid the system from staying converged at either $s = 1$ or $s = 0$. However, our experimental implementation below does not use such forced state changes.

Experimental Results: We give simulation data of the continuous Multiplicative protocol, 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 r_p is set to be the same at each process p. Figure 2 plots both the number of processes in state x, as well as the time until the last token is consumed in each round.

Figure 2(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 towards $x = 0$ to a convergence towards $x = 1$. Also, when $r_{avg} > 1$, not all tokens are consumed since each round becomes the maximum 200 periods, but this does not affect convergence. Figure 2(b) shows that massive failure of 50% of processes (at time $t = 100$) does not affect convergence.

To evaluate the effect of churn on the system, we injected the availability traces derived from the Overnet p2p system [Bhagwan et al. 2004] into a system running the Multiplicative protocol. Figure 3(a) shows the effect of injecting churn starting from time $t = 20$ onwards. While r_{avg} stays $= 1.05$ before $t = 80$, the system stays converged with all processes in state x. However, just as soon as r_{avg} becomes 0.95 at $t = 80$, the system quickly converges to all processes outside state x. Finally, Figure 3(b) shows that the protocol shows the desired convergence behavior even as r_{avg} changes intermittently (and before convergence). Notice that after each of these changes (at times $t = 50, 100, 150, 200$), the system adapts appropriately.

3.2 The Logistic Protocol as Interval Trigger Detector

The Logistic protocol is derived from the logistic sequence equation:

$$x_{m+1} = r \cdot x_m \cdot (1 - x_m), \text{with} \quad r \geq 0, 0 \leq x_i \leq 1 \tag{2}$$

This equation was first used by the Belgian sociologist and mathematician P. F. Verhulst to model the growth of populations [Strogatz 2001]. The Logistic protocol is derived from this equation by using the translation technique of Section 2 for polynomial multiplicative terms (Section 2.2.1.3). Each process maintains one state variable s, corresponding to x. Each process p checks at the beginning of a round whether it is in state s. If it is, a target process is selected uniformly at random, and its state queried (this is repeated if the target does not respond). Only if the target is *not* in state s, does process p generate an average r tokens for that round. These tokens are relayed and applied in the usual manner.

Equilibrium occurs when $s_{m+1} = r \cdot s_m \cdot (1 - s_m)$, i.e., $s = r \cdot s \cdot (1 - s)$. This has

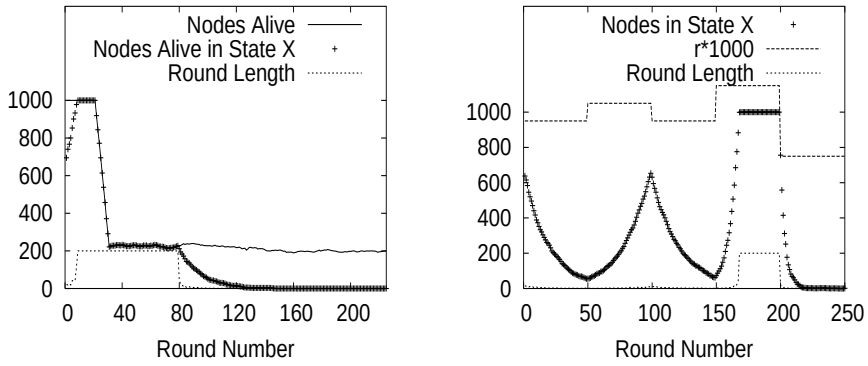


Fig. 3. **Multiplicative Protocol: Churned System and Varying $r = r_{avg}$.** (a) System begins to churn at $t = 20$, and average number of processes in churned system is about 200. Host availability traces from the Overnet p2p system are used. The value of r_{avg} changes from 1.05 to 0.95 at $t = 80$. Convergence occurs at $t = 147$. (b) The protocol reacts as $r = r_{avg}$ is varied around the threshold value. $N = 1000$.

two solutions: either $s = 0$ or $s = 1 - 1/r$. Notice that $\forall r$, $s = 0$ is an equilibrium, while $s = 1 - 1/r$ is an equilibrium only if $r \geq 1$.

To determine the stability of these points, we apply Lorenz's condition. Since $(r \cdot s \cdot (1 - s))' = r \cdot (1 - 2s)$, we have $f'(0) = r$. Thus the origin is stable for $r < 1$ but is unstable for $r > 1$. On the other hand, $f'(1 - 1/r) = 2 - r$, making this point stable iff $|2 - r| < 1$, or $1 < r < 3$.

Average Interval Trigger Detection: Similar to Section 3.1, we use a modified Logistic protocol for implementing the interval trigger. Here, each process uses a value of $r_p = \frac{R_p}{thresh}$ instead of parameter r while executing the actions of the Logistic protocol. The same arguments as Section 3.1 can then be used to show that the round to round behavior of this modified Logistic protocol is the same as the logistic equation with $r = r_{avg}$ at each process. Notice that the modified multiplicative protocol is in fact an extension of, and subsumes, the pure multiplicative protocol (the pure protocol can be derived when one sets $r_p = r$ at all processes).

In studying the behavior of the Logistic protocol, we will use the analysis from both [MathWorld 2007; Strogatz 2001]. First, as r is increased from 0 onwards, the only stable point is $s = 0$. As r crosses 1, the origin loses its stability; instead, a new stable point is created at $s = (1 - 1/r)$. We call this a *sensitive* bifurcation, because an equilibrium point jumps significantly [Strogatz 2001]. As such, it can be used to detect when the value of r used in the system crosses a lower threshold value of 1. On the other hand, as r continues to increase, at a value of $r = 3.0$, the stable point $(1 - 1/r)$ ceases to exist, and is instead converted into a *cycle* of stable points. In fact, it can be shown that for all values of $r > 3.0$, there is a stable cycle with at least two points [Strogatz 2001]. For all $r > 3$, the two points $p, q = \frac{r+1 \pm \sqrt{(r-3)(r+1)}}{2r}$ are such that $f(p) = q$ and $f(q) = p$. We call this an *insensitive* type of bifurcation because an equilibrium point does not jump but

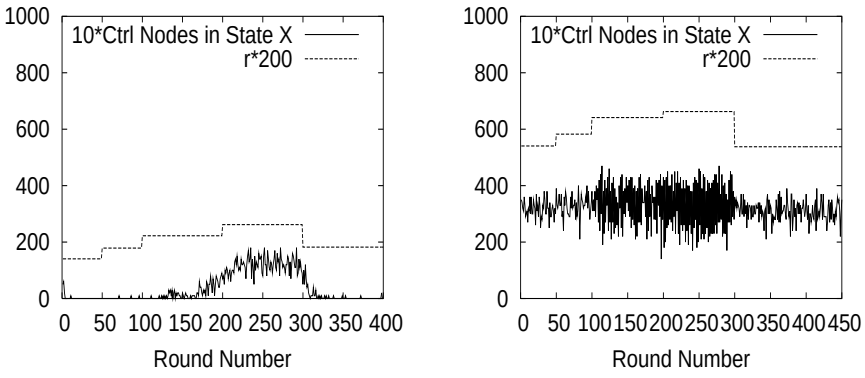


Fig. 4. **Logistic Protocol.** A control group of 50 processes is used. $N = 1000$. (a) Lower interval boundary. As r_{avg} crosses above a value of 1, the system has a non-zero stable number of processes in state s, while below 1, this number quickly drops to zero. (b) Upper interval boundary. As r_{avg} stays below a value of 3.0, the system is comparatively more stable than when > 3.0.

instead transitions smoothly. Thus, it is a little more difficult to detect the phase transition (although still possible)[4]. This bifurcation can be used to detect when r crosses the upper threshold of 3.0.

Thus, the Logistic protocol can be used to detect a global interval trigger by piecewise-linear mapping the range $[0, R_{max}]$ of the parameter of interest R into the range of $r \in [0, 3.5]$, so that the trigger range to be detected is mapped to the interval $r \in [1.0, 3.0]$. We do not consider $r > 3.5$ as it is chaotic.

Convergence Times: We analyze the convergence times (in rounds) only for $0 < r < 3$, ignoring $r > 3$ because of the cyclic nature of that range. For $r < 1$, with N processes initially in state s, the expected number after t rounds is upper-bounded by $N \cdot r^t$. The expected convergence time is $O(\log_{1/r}(N))$ rounds.

For $r \in [1, 3]$, some values of r have closed form solutions [MathWorld 2007]. For example, at $r = 2$, the solution is $s_m = \frac{1-e^{-2^n \cdot c}}{2}$, where c is some constant [MathWorld 2007], giving a convergence time of $O(\log(\log(N))$. Other r values may have longer convergence times, but this time is always $O(N)$ since s_m is monotonic with m.

The Logistic protocol can be executed not only in a one-shot mode as described, but also in a continuous mode. The latter variant requires periodic forcing of a few processes into state s, and a few others out of state s (neither overlapping with the control group). This is to avoid the system from becoming "stuck" at an equilibrium. Our implementation below uses the continuous version, but does not rely on such forced state changes.

Experimental Results: The experimental conditions for the continuous Logistic

[4]This is also called a *flip* bifurcation [Strogatz 2001]. Further, the logistic equation displays chaotic behavior at values of $r > 3.5$, however we are unable to harness the use of this behavior for practical purposes at this time.

protocol are the same as in Section 3.1. A small control group of 50 processes is used. Figure 4(a) and (b) show the respective behavior when r_{avg} is near the lower and upper boundaries of the interval $[1, 3]$. Each process p sets r_p to a randomly-selected point in the interval $[max(0, r_{avg} - 0.5), min(3.5, r_{avg} + 0.5)]$.

Figure 4(a) shows that as r crosses above 1.0 (the lower end of the trigger interval), a significant fraction of the control processes move into state s, e.g., between times $t = 200$ and $t = 300$. On the other hand, when r crosses below 1.0, almost all control processes are out of state s, e.g., between times $t = 300$ and $t = 400$. Thus, transition at the lower end of the trigger interval can be detected.

Figure 4(b) shows that as r crosses above 3.0 (the upper end of the trigger interval), the insensitive nature of the bifurcation causes an increased perturbation (over time) in the number of control processes in state s, e.g., between times $t = 200$ and $t = 300$. Thus, transition at this upper end of the trigger interval can be detected if the number of control processes in state s varies highly over several rounds [5]. Massive failure and churn did not appear to affect convergence (just as in Section 3.1), and are thus excluded.

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 *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" [Herbster and Warmuth 1995]. However, those are exhaustive and potentially expensive if applied to a p2p distributed system. Furthermore, Rice showed that it is prohibitively expensive to find the best algorithm even if one knows the input data set characteristics [Rice 1976].

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 [D'Silva 1998; Seeley 1996]. 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

[5] Notice that this insensitive bifurcation is more challenging to detect than the sensitive detection at $r = 1.0$.

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 [Arpaci-Dusseau et al. 1998]. HoneyAdapt can also be easily optimized for specific applications, e.g., by combining it with machine-learning techniques [Guo 2003]; 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 component 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 [D'Silva 1998] that model the actual honeybee behavior:

$$a_i\,(t) = \; a_i\,(t-1) \cdot (1 - pf_i) + \left(\frac{sq_i \cdot pd_j \cdot a_i\,(t-1)}{\sum_{j=1}^{A} sq_j \cdot pd_j \cdot a_j\,(t-1)} \right) \cdot \sum_{j=1}^{A} pf_j \cdot a_j(t-1) \tag{3}$$

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 (3) 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 (3)). 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 (3)).

4.2 Analysis of the HoneyAdapt Protocol

The discussion in Section 2.1 hypothesized that HoneyAdapt's emergent behavior can be probabilistically predicted by the equation (3), and Section 3 provided empirical evidence of it being true. Below, assuming this hypothesis is true, we analyze the convergence behavior of HoneyAdapt by analyzing the source sequence equation. 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 (3).

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 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 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 Lemma 2 in the Appendix): in a sequence equation with multiple variables, the stability condition 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 1, 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 three additional strategies defined by {memory vs. memoryless} × {greedy vs. sampled}.

We use a network topology generated via BRITE's [Medina et al. 2001] 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 II 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 run-time, 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.

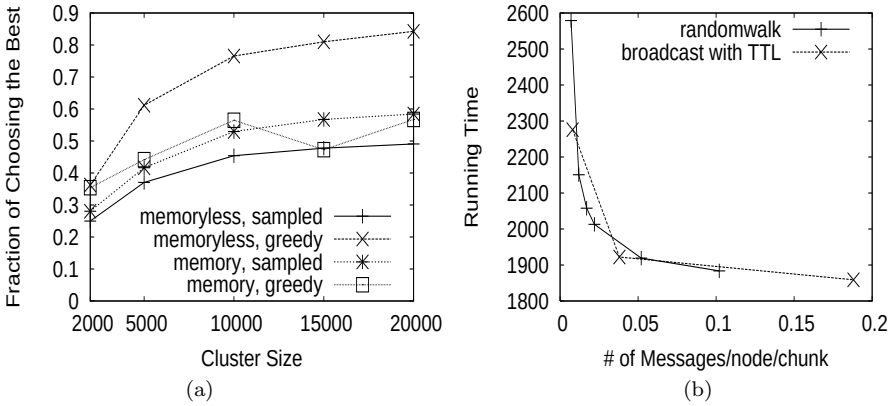
Our main metrics are: (1) % chunks that end up being executed with their individual fastest component algorithms, and (2) running time for the entire data set, calculated as sum of execution times for all chunks across all clients.

Algorithm Variations: We evaluate four variants of HoneyAdapt, as defined by {memory vs. memoryless} × {greedy vs. sampled}. Recall that the original

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

Table II. **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



(a) (b)

Fig. 5. **HoneyAdapt Variants.** (a) Algorithm selection: {memory vs. memoryless} × {greedy vs. sampled}. (b) Message propagation: randomwalk vs. broadcast with TTL.

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 5(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 5(b) shows a comparison of the message-overhead tradeoff between two techniques to spread advertisement messages in the overlay. These techniques are - a random walk and a broadcast, both with a TTL (=time-to-live). Different plot points are derived by varying the value of TTL. The x axis plots the normalized message overhead per node, per chunk processed by it. In general, as the message overhead increases, we observe that the running time plateaus out. Thus there is not much benefit in spending more than 0.05 messages per node

per chunk. Further, since the two algorithms define similar tradeoff boundaries, hence we use a random walk from now on.

Cluster Size: Figure 6 varies the number of consecutive chunks of the 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., 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.

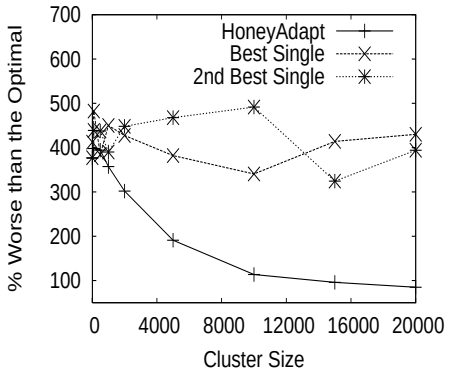


Fig. 6. **Effect of various cluster sizes.**

Scalability: Figure 7 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 $TTL = \lceil \frac{N}{500} \rceil$. The plot shows that the bandwidth and relative running time of HoneyAdapt both improve with N. Notice that the bandwidth is high for 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

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

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

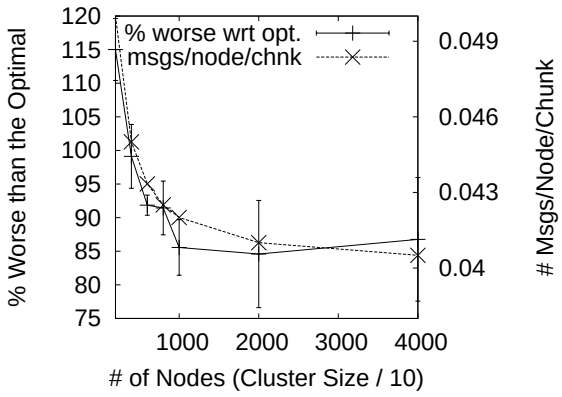


Fig. 7. **HoneyAdapt with various numbers of nodes.**

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. This eliminates the need for synchronizing rounds across processes, unlike the prior sequence protocols in this paper.

(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}). The advantage of using pr is that the system does not get "stuck", i.e., all processes converging to use only one component algorithm. The perturbation caused by the use of pr allows the system to converge to a better algorithm if the input set characteristics change.

(3) Advertisements are forwarded only to immediate overlay neighbors, and only the latest advertisement from each neighbor is used. This means that TTL=1, thus reducing the message overhead of the protocol.

We deployed HoneySort on a Linux PC cluster with a TCP-based overlay, which is a complete graph by default (see Table III for random overlays). Component algorithms are classical Quicksort and classical Insertion sort [Cormen et al. 2001]; randomized quicksort and mergesort yield similar results. Datasets consist of 8 Byte integer element input arrays created via SortGen [Gray 2005].

Figures 8(a) and (b) show running times respectively for completely random and pre-sorted input arrays. For random arrays, where Quicksort is faster than Insertion sort, Figure 8(a) shows that HoneySort adapts to perform as well as Quicksort. For mostly sorted arrays, where Insertion sort is faster than Quicksort, Figure 8(b) shows that HoneySort adapts to perform as well as Insertion sort.

Figure 9(a) shows data for heterogeneous arrays, where groups of *dynamic change unit* chunks are selected within the input array, and groups are alternately random

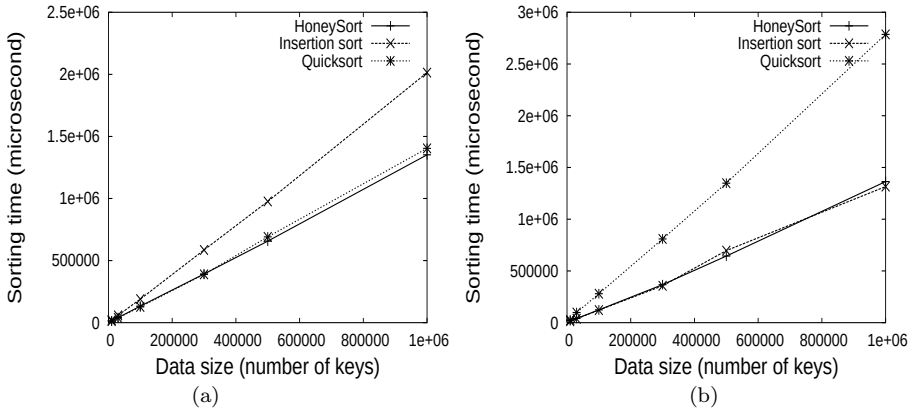


Fig. 8. **HoneySort:** (a) On Random input. With chunk size approx. 1000 and with 6 client machines; (b) On Pre-sorted input. With chunk size approx. 1000 and with 6 client machines.

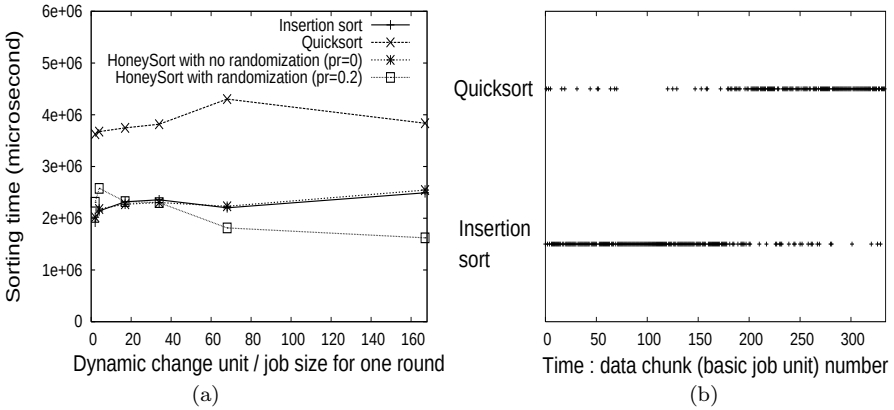


Fig. 9. **HoneySort on heterogeneous arrays.** (a) Input array has 1 million entries, chunk size approx. 3000 elements, 333 chunks total. With 6 client machines. Randomization prob. $pr = 0.2$. Groups of dynamic change unit (x-axis) chunks are alternately random and sorted. (b) A sliver showing HoneySort's adaptivity - the first 150 chunks are each sorted, while the last 150 are each 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.* Figure 9(b) demonstrates that HoneySort is able to adapt quickly even as the nature of incoming chunks changes dynamically. This would make HoneySort appropriate for online computation as well.

Finally, Table III shows that even for sparse overlays (lower degree), the running times fall by only about 35%. This is partly because the degree affects only *pd*, which in turn does not affect protocol behavior.

Table III. **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.

Node Degree in Overlay	HoneySortRunning Time (ms)
2	945183
8	790736
15	1086741
29	882870

6. SUMMARY

This paper has shown how 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, Logistic protocol) or enable clients in a large-scale Grid application to adapt at run-time to execute large data-sets more efficiently (HoneyAdapt). HoneySort, an adaptive parallel sorting protocol, outperformed well-known algorithms Insertion Sort and Quicksort.

We are hopeful that the design methodology described in this paper can be applied by researchers to other well-known sequence equations in literature [Agarwal 2000; Strogatz 2001], in order to create useful and practical distributed protocols. Our successful case studies in this paper (Multiplicative protocol, Logistic protocol, HoneyAdapt and HoneySort) lead us to hypothesize that this might be a fruitful direction.

At the same time, we wish to point out that our design methodology does not completely automate the creative process of protocol design. In other words, given a distributed computing problem, the challenge of identifying the appropriate natural phenomena that is a good solution match, still remains an open direction. Thus, another future direction that this paper opens up is the possibility of using sequence equations as a specification language for distributed computing problems and desired protocol properties, thus automatically producing distributed protocol solutions.

Acknowledgments: We wish to thank Imranul Hoque for his detailed and insightful comments on the paper.

REFERENCES

Agarwal, R. P. 2000. *Difference equations and inequalities: theory, methods and applications*, 2 ed. CRC.

Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M. J., and Peralta, R. 2006. Computation in networks of passively mobile finite-state sensors. *Distributed Computing 18,* 4, 235–253.

Arpaci-Dusseau, R., Arpaci-Dusseau, A., Culler, D. E., Hellerstein, J. M., and Patterson, D. A. 1998. Searching for the sorting record: experience with now-sort. In *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM, New York, 124–133.

Arvind. 2003. Bluespec: A language for hardware design, simulation, synthesis and verification. In *Proceedings of ACM-IEEE MEMOCODE*. ACM, New York, 249.

Babaoglu, O., Jelasity, M., Canright, G., Urnes, T., Deutsch, A., Ganguly, N., di Caro, G., Ducatelle, F., Gambardella, L. M., Montemanni, R., and Montresor, A. 2005. Design patterns from biology for distributed computing. In *Proceedings of the European Conference on Complex Systems*.

Babaoglu, O., Meling, H., and Montresor, A. 2002. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, New York, 15.

Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S., and Voelker, G. M. 2004. Total Recall: System support for automated availability management. In *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*. Usenix, San Francisco, CA, 337–350.

Bonabeau, E., Dorigo, M., and Theraulaz, G. 1999. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Oxford, Great Britain.

Chandy, K. and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3*, 63–75.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. 2001. *Introduction to Algorithms*, 2 ed. MIT Press, Cambridge, MA.

di Caro, G. and Dorigo, M. 1998. Antnet: Distributed stigmergic control for communications networks. *Journal of AI Research 9*, 317–365.

D'Silva, S. 1998. Collective decision making in honeybees. http://www.msu.edu/user/dsilva/beepp2.ps.

Gray, J. 2005. Sorting benchmark. http://research.microsoft.com/barc/SortBenchmark.

Guo, H. 2003. Algorithm selection for sorting and probabilistic inference: a machine-learning approach.

Gupta, I., Nagda, M., and Devaraj, C. F. 2007. The design of novel distributed protocols from differential equations. *Distributed Computing 20,* 2, 95–114.

Herbster, M. and Warmuth, M. K. 1995. Tracking the best expert. In *Proceedings of International Conference on Machine Learning (ICML)*. ACM, New York, 286–294.

Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and Steen, M. 2007. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS) 25,* 3 (August), Article 8.

Kempe, D., Dobra, A., and Gehrke, J. 2003. Computing aggregate information using gossip. In *Proc. 44th IEEE Symposiumon Foundations of Computer Science (FOCS)*. IEEE, New York, 482–491.

Ko, S. Y., Gupta, I., and Jo, Y. 2007. Novel mathematics-inspired algorithms for self-adaptive peer-to-peer computing. In *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, New York, 3–12.

Kurtz, T. 1981. Approximation of population processes. In *Regional Conference Series in Applied Mathematics (SIAM)*. SIAM, Philadelphia, PA.

Loo, B., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. 2005. Implementing declarative overlays. *ACM SIGOPS Operating System Review 39,* 5, 75–90.

Maniezzo, V., Boschetti, M. A., and Jelasity, M. 2004. An ant approach to membership overlay design. In *Proceedings of the International Workshop on Ant Colony Optimization and Swarm Intelligence(ANTS), pages=37-48, publisher (Springer LNCS 3172*. Berlin, Germany.

MathWorld. 2007. Logistic map. http://mathworld.wolfram.com/LogisticMap.html.

Medina, A., Lakhina, A., Matta, I., and Byers, J. 2001. BRITE: An approach to universal topology generation. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*. IEEE, New York, 346.

Merritt, M. and Taubenfeld, G. 2000. Computing with infinitely many processes. In *Proc. International Conference on Distributed Computing (DISC)*. Springer LNCS 1914, Berlin, Germany, 164–178.

Misra, J. and Chandy, K. M. 1982. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4,* 1 (January), 37–43.

Mitzenmacher, M. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 12,* 10 (Oct.), 1094–1104.

Rabin, M. O. 1983. Randomized Byzantine generals. In *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, New York, 403–409.

Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers 15,* 65, 118.

Rohrer, J. 2007. Mute: Simple anonymous file sharing. http://mute-net.sourceforge.net/.

Schneider, F. 1986. The state machine approach: A tutorial. Tech. Rep. TR86-800, Cornell University.

Seeley, T. 1996. *The Wisdom of the Hive*. Harvard University Press, Harvard, MA.

Strogatz, S. H. 2001. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*, 1st ed. Perseus Book Group, New York.

Uresin, A. and Dubois, M. 1990. Parallel asynchronous algorithms for discrete data. *Journal of the ACM 37,* 3 (July), 588 – 606.

Voulgaris, S., Gavidia, D., and van Steen, M. 2005. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management 13,* 2 (June), 197–217.

Appendix: Extension of Lorenz's Stability Condition

LEMMA 2. *(Extension of Lorenz's stability condition to multi-variable sequence equations) In sequence equation with multiple variables* $\vec{x}(t) = \vec{H}(\vec{x}(t-1))$*, the stability condition at an equilibrium point* $\vec{x}(\infty)$ *is that all eigenvalues of the Jacobian* $\begin{pmatrix} \frac{\partial H_1}{\partial x_1} & \frac{\partial H_1}{\partial x_2} & \cdots \\ \frac{\partial H_2}{\partial x_1} & \frac{\partial H_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$ *at the equilibrium point should have absolute values strictly less than* 1*.*

PROOF. The stability condition at an equilibrium point $\vec{x}(\infty)$ requires that if we introduce a small perturbation from the equilibrium point at time t, $\vec{x}(t) = \vec{x}(\infty) + \vec{\delta}$, its next point $\vec{x}(t+1) = \vec{H}\left(\vec{x}(\infty) + \vec{\delta}\right)$, is closer to the equilibrium point than $\vec{x}(t)$ is to the equilibrium point. That is, $|\vec{x}(t+1) - \vec{x}(\infty)| < |\vec{x}(t) - \vec{x}(\infty)| = |\vec{\delta}|$ for an arbitrary small $\vec{\delta}$. This gives us the following derivation:

$$\begin{aligned}
& \begin{pmatrix} x_1(t+1) \\ x_2(t+1) \\ \vdots \\ x_m(t+1) \end{pmatrix} \\
= & \begin{pmatrix} H_1\left(\vec{x}(\infty) + \vec{\delta}\right) \\ H_2\left(\vec{x}(\infty) + \vec{\delta}\right) \\ \vdots \\ H_n\left(\vec{x}(\infty) + \vec{\delta}\right) \end{pmatrix} \\
\simeq & \begin{pmatrix} H_1(\vec{x}(\infty)) + \frac{\partial H_1}{\partial x_1}\delta_1 + \ldots + \frac{\partial H_1}{\partial x_m}\delta_n \\ \vdots \\ H_n(\vec{x}(\infty)) + \frac{\partial H_n}{\partial x_1}\delta_1 + \ldots + \frac{\partial H_n}{\partial x_m}\delta_n \end{pmatrix} \\
= & \begin{pmatrix} \frac{\partial H_1}{\partial x_1} & \frac{\partial H_1}{\partial x_2} & \cdots \\ \frac{\partial H_2}{\partial x_1} & \frac{\partial H_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} + \begin{pmatrix} x_1(\infty) \\ \vdots \\ x_m(\infty) \end{pmatrix}
\end{aligned}$$

The above condition is equivalent to saying:

$$\vec{x}(t+1) - \vec{x}(\infty) = \begin{pmatrix} \frac{\partial H_1}{\partial x_1} & \frac{\partial H_1}{\partial x_2} & \cdots \\ \frac{\partial H_2}{\partial x_1} & \frac{\partial H_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} \tag{4}$$

By the stability requirement,

$$\left| \begin{pmatrix} \frac{\partial H_1}{\partial x_1} & \frac{\partial H_1}{\partial x_2} & \cdots \\ \frac{\partial H_2}{\partial x_1} & \frac{\partial H_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} \right| < \left| \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} \right|$$

Suppose the Jacobian has eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$ and corresponding eigenvectors $V_1, V_2, \ldots, V_m$. Suppose $\vec{\delta} = c_1 \cdot V_1 + c_2 \cdot V_2 + \ldots + c_m \cdot V_m$, where all c_i's are constants. The above inequality can then be shown to reduce to:

$$\sum_{i=1}^{m} c_i^2 (1 - \lambda_i^2) > 0$$

Now, notice that if $\forall i, |\lambda_i| < 1$, then the above inequality is true. For the contrapositive, if there is some $j : |\lambda_j| \geq 1$, then choosing $\vec{\delta}$ such that $c_j > 0$ and $\forall i \neq j : c_i = 0$, violates the above inequality. Hence proved. □

Received September 2007; revised January 2008; accepted May 2008