On the Use of Sequences, Phase Changes, and HoneyBees For Designing Adaptive Distributed Systems*

Indranil Gupta and Yookyung Jo
Dept. of Computer Science
University of Illinois at Urbana-Champaign, IL 61801.
{indy,yjo}@cs.uiuc.edu

Abstract

The invention of distributed protocols is an extremely challenging activity. Today however, few design paradigms are available for the creative task of designing scalable and reliable protocols for the Grid, peer to peer systems, etc. This paper first presents a **design methodology** *to translate sequence equations, that are extensions of the form* $x_{m+1} = f(x_m, x_{m-1}, \ldots, x_{m-k})$*, into distributed protocols that are provably equivalent, i.e., exhibit the same equilibrium points and trajectories. These novel* **sequence protocols** *are decentralized, simple, scalable, and highly fault-tolerant. We then demonstrate how phase changes in sequence protocols can be used to detect certain global predicates in a decentralized manner. Two such new protocols called the* **Multiplicative protocol** *and the* **Logistic protocol** *are presented, rigorously analyzed, and experimentally studied. Finally, we present* **HoneySort**, *a novel sequence protocol for adaptive Grid computing. HoneySort is derived from sequence equations representing coordination among honeybees in nature. Through rigorous analysis and experiments with a real deployment on a 30-node PC cluster, we show that HoneySort outperforms well-known distributed sorting algorithms such as Quicksort and Insertion sort.*

Keywords: *Science of Protocol Design, Distributed Protocols, Sequence Protocols, Scalability, Reliability.*

1 Introduction

The invention of distributed protocols that are simple, scalable and robust, is crucial to the success of large-scale distributed technologies such as the Grid, distributed storage, and peer-to-peer systems. Yet, surprisingly few paradigms are available today to assist a protocol designer in this creative activity.

In this paper, we present an innovative design methodology that translates certain types of *sequence equations* into new distributed protocols. The translated protocol exhibits equivalent behavior to the original equation. The canonical single-variable sequence equation is of the form $x_{m+1} = f(x_m, x_{m-1}, \ldots, x_{m-k})$, where k is a constant integer, f is a well-known function with a finite number of terms, m takes on integer values $\geq k$, and all x_i's take real values $\in [0, 1]$. We also extend our methodology to multi-variable sequence equations.

We call the protocols generated from sequence equations as *Sequence Protocols*. Sequence protocols are completely decentralized, simple to express, scalable since they involve *constant* per-node communication overheads and memory, and are highly fault-tolerant due to their stochastic nature. A sequence protocol operates in *rounds*. The round duration is fixed, and is typically at least several minutes long. We assume that all processes can start each round at the same time. While such coarse granularity synchronization can be provided by the widely available Network Time Protocol (NTP), in practice, this round synchrony assumption can be completely relaxed.

A sequence protocol has the same trajectories and equilibrium points as the source sequence equation. Since sequence equations are already used to model natural phenomena, our methodology allows us to design distributed protocols that inherit many important properties such as self-stabilization and phase changes, and to use these phenomena for building adaptive distributed systems.

We demonstrate the practicality of this design approach by studying three new protocols generated by the methodology. The *Multiplicative protocol* and the *Logistic protocol* use phase change behavior to detect, in a decentralized manner, violations of certain global predicates. Such a diagnostic capability is vital to building adaptive distributed systems. The third protocol translates sequence equations for a model of coordination among honey-bees into an adaptive Grid computing protocol. This leads to a new adaptive distributed sorting protocol called *HoneySort*. Our experiments show, rather surprisingly, that *HoneySort outperforms distributed versions of Quicksort and Insertion sort.*

*This work was partially supported by National Science Foundation Grant ITR-0427089.

Related Work: Previously in [6], we translated *continuous* differential equations into distributed protocols. Unfortunately, those techniques are inapplicable for translating sequence equations as these are *discrete*. Further, a sequence equation typically has more pronounced phase change behavior than its continuous counterpart, making it much more interesting.

Population protocols [1, 9] 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 [7]. Methodologies in general have been used to systematize the design process in many fields, e.g., [2], but in distributed computing, they have begun to emerge only recently, e.g., [8, 13]. Yet, most of these methodologies tend to compose existing protocols, rather than invent new protocol classes. Sequence protocols are quite different from checkpointing and virtual synchrony protocols; the latter two may in fact be unscalable. Many other protocols also operate in rounds, e.g., [10]; to our knowledge, none of these can be generated from sequence equations.

System Model: For simplicity of analysis, we assume a closed group of processes, N of which are non-faulty. They communicate over a reliable network. N is assumed to be very large. Reliable communication can be provided by TCP channels. In practice, all these assumptions can be relaxed – our experiments show that sequence protocols are resilient to massive failures and churn failures, in finite sized groups. All processes are also assumed to know exactly when each round begins – as mentioned before, coarse granularity synchronization provided by NTP suffices. This assumption can also be relaxed by having each process start the next round when either a local timer expires, or it sees a message from some other process for the new round. Our HoneySort protocols works without using rounds at all, and is fully asynchronous. Finally, we assume that each process also knows about the maximal group membership, i.e., the other $N-1$ processes. For dynamic groups, this can be relaxed by a membership protocol at each process.

Section 2 describes the translation methodology. The Multiplicative protocol and Logistic protocol are presented in Section 3. Section 4 details the HoneySort paradigm. We conclude in Section 5.

2 A Methodology to Generate Sequence Protocols

First, we consider how to translate single-variable sequence equations only:

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

As described in Section 1, k is a constant and all $x_i \in [0, 1]$. Later, we extend our techniques to equations with multiple variables. Suppose f is written as a sum of a finite number of elementary *terms* (positive or negative).

The translation works by converting (i) the equation variable into a state variable for the protocol, and (ii) each term in the function f into a set of protocol actions. For the variable x, a local state variable is defined at each process - this takes on boolean values, indicating whether the process is in that state or not. For process p, we call this as the "state variable x at p". When the state variable at process p is 1, we say that "p is in state x"; when the state variable is 0, we say that "p is out of state x". Thus, the value of x in the sequence equations translates to the fraction of processes in state x.

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

The derived sequence protocols are equivalent to the sequence equations in the sense that for each variable x_i, the *fraction* of processes that are in state x varies over time exactly as they would in the sequence equations.

Example: An example of a sequence equation that satisfies the above restrictions is the *multiplicative map*:

$$x_{m+1} = r.x_m \quad (2)$$

where r is a positive constant. The value of k is thus 0, and f consists of one polynomial term. We will use this sequence equation as a working example.

2.1 Basic Translation Methodology

The basic generated protocol is completely decentralized. At each process, the boolean state variable x is updated at the start of each round. In addition, the process always remembers the immediately previous values of x, i.e., at the start of the previous k rounds. Finally, the process maintains a variable x_{next} that is a running estimate of its state variable for the start of the next round; x_{next} is continuously updated during the current round.

At start of a given round at process p, the value of the boolean state variable is set to to x_{next}, and p initializes $x_{next} := 0$. Then, process p executes two types of actions – *Token Generation*, followed by *Token Relay and Apply*. Token generation creates a number of *token* messages, based on the terms in the sequence equations. Token messages are then *relayed* to other processes through

random walks. In turn, when p receives tokens, it *applies* these tokens to x_{next}. At the next round start, besides updating the state variable to x_{next}, process p also updates its list of previous k remembered state variable values.

This generic sequence protocol is described in Figure 1. We elaborate on the two main actions below:

1. Token Generation: Each generated token can be either *positive* or *negative*. Positive (resp. negative) tokens are generated for each positive (resp. negative) term in f. The goal of the token generation procedure is to have the N processes create, in a decentralized manner, per round, $T \times N$ positive (resp. negative) tokens for each positive (resp. negative) term T.

2. Token Relay and Apply: When a process with $x_{next} == 0$ receives a positive token, it consumes the token and sets x_{next} to 1. When a process with $x_{next} == 1$ receives a negative token, it consumes the token and sets x_{next} to 0. Otherwise, the process forwards the token to a random non-faulty target process. Thus, each token takes a random walk until it is consumed.

Although a process has to execute token generation at the start of the round, the relay and apply operations are completely asynchronous across processes in the system. Thus, no synchrony is required within the round.

Lemma 1: *Consider a single-variable sequence equation. Suppose the sequence protocol is such that, per round: (i) for each term $+T$ (resp. $-T$) on the right hand side of the sequence equation, the number of positive (resp. negative) tokens generated is $T \times N$, and (ii) the round duration is long enough for all tokens generated by the algorithm of Figure 1 to be consumed. Then Figure 1 is a protocol that has equivalent behavior to the original single-variable sequence equation.*

2.2 Token Generation

We describe below how token generation works for different types of terms, ignoring the positiveness or negativeness of the terms. Recall that token generation actions are decentralized and aim to create, for each term T, $T \times N$ tokens per round, in the system.

2.2.1 Polynomial Terms

Constant term of form $T = r$, where r is a constant: Each process (regardless of its state at the start of the round) *generates an average of r tokens*. This is achieved by generating $\lfloor r \rfloor$ tokens, and then generating an extra token with probability $(r - (\lfloor r \rfloor))$. These tokens are all positive if $r > 0$, otherwise they are all negative. This token generation action requires no message exchange.

Polynomial terms of form $T = r.x_{m-j}$, where $j \leq k$: A process checks if it was in state x at the start of j

```
boolean x_last; // state at start of this round
boolean x_{last-1}, ... x_{last-k}; // states at start of last k rounds
boolean x_next; // next state - running variable

Round m:
int numtokens_pos, numtokens_neg, numtokens;
    //number of tokens: positive, negative, net

    for (i = k down to 1)
        set x_{last-i} := x_{last-i+1}; // remember the last k states

    set x_last := x_next;
    set x_next := 0;

    Use Token Generation Algorithm(f, x_last, ... x_{last-k})
    to generate tokens;
        let numtokens_pos and numtokens_neg respectively be the
              number of positive and negative tokens generated locally;
        set numtokens := numtokens_pos - numtokens_neg;
    if (numtokens > 0)
        set x_next := 1
              at the start of next round
        numtokens := numtokens - 1;

    if (numtokens ≠ 0)
        contact |numtokens| distinct non-faulty target processes,
        each chosen uniformly at random from the group (retrying if
        target is unresponsive);
              if (numtokens > 0)
                  send each target process a message containing one
                  positive token;
              else
                  send each target process a message containing one
                  negative token;

//Token Relay and Apply
    while (round is not over)
        if (receive message with a positive token)
              if (x_next==1)
                  select one distinct non-faulty target process,
                        chosen uniformly at random from the group
                        (retry if target unresponsive);
                  send to the target process a message containing
                  one positive token;
              else
                  set x_next := 1; // consume token
        else// if(receive message with a negative token)
              if (x_next==0)
                  select one distinct non-faulty target process,
                  chosen uniformly at random from the group
                        (retry if target unresponsive);
                  send to the target process a message containing
                  one negative token;
              else //if(x_next==1)
                  set x_next := 0; // consume token
```

Figure 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).

rounds ago – if yes, it generates an average of r tokens, in a similar manner as above. This token generation action requires no message exchange.

Example: The multiplicative map $x_{m+1} = r.x_m$ can be translated using this token generation rule. Figure 13 in Appendix A depicts the Multiplicative protocol, an instantiation of Figure 1 using the above token generation rule. Section 3.1 will detail use of the Multiplicative protocol for detecting global predicates.

Polynomial terms of form $T = r.\Pi_{i=m}^{i=m-k} x_i^{(j_i)}$ (each j_i a non-negative integer, some j_i positive): This term is a product of several variables. Token generation here requires the process to sample the group through messages. Let i' be the highest value of i in the term T such that $j_i > 0$. In the protocol, at the start of a round, a process p first checks if it was in state x at the start of $(m - i')$ rounds ago. If not, it takes no action. If yes, it sends out $(\sum_{i=m}^{i=m-k} j_i) - 1$ sampling messages, each to a target process chosen uniformly at random. For any process that is non-responsive, a different random target is retried. Non-faulty target processes reply immediately with the list of states they were in for the last k rounds. Process p then checks if 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 $T/r.x_{i'} = x_{i'}^{(j_{i'}-1)}.\Pi_{i=i'-1}^{i=m-k} x_i^{(j_i)}$, when the individual variables of the product are arranged in lexicographically decreasing order[1]. If this condition is true, process p generates an average of r tokens for that round.

Ex: If $T = 3.2 \times x_m^2.x_{m-3}^2$, then $i' = m$, $r = 3.2$, and $T/r.x_{i'}$ is written as $x_m.x_{m-3}.x_{m-3}$. Each process p in state x at the round start sends out 3 sampling messages. If the first received reply is from a process in state x, and the remaining two replies are from processes that were in state x at the start of 3 rounds ago, then an average of 3.2 tokens are generated by p for this round.

Translating Sequence Equations with Multiple Variables: The above methodology can be easily generalized to translate a system of sequence equations. A system of sequence equations with $l > 1$ variables $\{x1, x2, \ldots, xl\}$ specifies a sequence equation for each $xi (1 \leq i \leq l)$: $xi_{m+1} = f_i(xj_b, 1 \leq j \leq l, m-k \leq b \leq m)$. Further, f_i is finite, k is a constant integer, and $\forall j, b : xj_b \in [0, 1]$. In the derived protocol, for each i $(1 \leq i \leq l)$, each process maintains one state variable xi (and a memory of the last k values of it). Separate token generation actions are created from each f_i and executed at each round. The tokens generated from f_i are tagged with i so that they are applied only to state variables xi. Note that if f_i contains terms with xj_b (for some b, $j \neq i$), then the token generation actions related to state variable xi generalize to also involve or sample values of state variables xj ($j \neq i$), either locally or sampled remotely.

Lemma 2: *For a term T that is either constant or polynomial, the actions of the above methodology generate $T \times N$ tokens at the start of each round.*

By applying Lemma 2 to an obvious extension of Lemma 1, we have:

Theorem 1: *Given a system of sequence equations with a finite number of terms, and where all variable values $\in [0, 1]$, and in which all terms T are either constant or polynomial, the above methodology generates a protocol that is equivalent in behavior, i.e., in trajectories as well as equilibria.*

2.2.2 Non-Polynomial Terms

First, notice that some non-polynomial terms can be translated by approximating them with a truncated sum of polynomials and then translating, e.g., $e^x \simeq 1 + x + x^2/2 + x^3/6$, and $\frac{1}{1-x} \simeq 1 + x + x^2$.

Below, we describe *exact* translation for some non-polynomial terms. Exact translation of a term requires each round to be split into a known number of *subrounds*, and the use of *subtoken* messages. Just like for rounds, each process knows when each subround starts. Subtokens do not affect state variables directly, but instead contribute towards generation of tokens. The actions for individual terms are tied together by using the maximum subrounds across all terms, as the number of subrounds in the final protocol. We ignore such tying issues below, focusing only on individual terms.

Division terms $T = r/x_m$: Split each round into two subrounds. In the first subround, each process p (regardless of its state at the round start) iteratively selects one target process uniformly at random (which replies with its state), until the target process is found to be in state x at this round start. If a total of z targets were responsive, then p generates an average $r.z$ tokens. In the second subround, the tokens are relayed and applied as usual. It is easy to see that the expected number of tokens generated is $N \times r.1/x_m$, as desired.

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

[1] In fact, any arbitrary ordering of the variables would suffice.

To translate this term, two subrounds are required per round. In the first subround at process p, for each i such that p is in state xi, p generates an average a_i subtokens. Each subtoken carries the value of i that generated it. Then, p multicasts these subtokens to *all* other processes in the group. Notice that p receives, for each i, $N \times a_i.xi$ subtokens carrying i. In the second subround, p selects one subtoken at random from those received. Only if the j value in the subtoken is such that b_j is 1 (in term T), does process p generate an average of r tokens.

The first subround multicasting can be relaxed by having processes spread subtokens only to their neighbors in an overlay. A process that has received no subtokens after the first subround needs to sample other processes to fetch subtokens.

Lemma 3: *For a term T that is either a division term, or a fractional term, the actions of the above methodology generate $T \times N$ tokens at the start of each round.*

Recursive Translation: Consider a translatable term T, and substitute a *subterm* occurring in T with an arbitrary function $g(x_m, \ldots, x_{m-k})$. If g has a range within $[0, 1]$, and g is translatable, then the resulting term T' can also be translated as follows. Split the round into two subrounds. Translate g to a first subround action that generates subtokens for g (totaling $N \times g(x_m, \ldots, x_{m-k})$), then relays the subtokens until each process has either 0 or 1 g subtokens. The second subround uses the translated action from T, except that the actions relating to the substituted subterm are replaced with a sampling of the g subtokens. Ex: Since $T = r/x_m$ is translatable, so is $T = r/(x_m^2)$.

Notice that this procedure is recursive. By combining Lemmas 1,2,3, and Theorem 1, we have the following theorem (which can be applied recursively):

Theorem 2: *Consider a sequence equation system with a finite number of terms, and all variable values $\in [0, 1]$, and where each term T is either: (i) constant, polynomial, division, or fractional, or (ii) can be rewritten into one of these forms by a finite set of recursive substitutions, of subterms within T, with dummy variables. Then the sequence protocol generated by the above methodology is equivalent to the equation system, i.e., in trajectories and equilibria.*

3 Using Phase Changes to Detect Global Predicates

Equilibria and Perturbations: All trajectories of a sequence protocol with l state variables $\{x1, x2, \ldots, xl\}$ lie in $[0, 1]^l$. Equilibrium points can be one of three types: stable, unstable, or saddle. All points in the neighborhood of a stable point (resp. unstable point) converge towards it (resp. diverge from it). For a saddle point, some points in the vicinity diverge while other points converge.

In practice, however, the behavior of the sequence protocol is affected by *perturbations* that naturally arise from imperfect sampling, imperfect random number generators, etc. Perturbations are beneficial because they do not affect the convergence around a stable point, but cause a system exactly at an unstable or saddle point to eventually diverge away by moving it into the unstable neighborhood.

Phase Changes for Global Predicate Detection: As one varies the *constants* involved in the original sequence equations, the location of equilibrium points moves. In addition, at critical values of the constants called *bifurcation points*, the number and/or types of the equilibrium points may change too. Such bifurcations are called *phase changes*.

The marked behavioral change at bifurcation points allows us to use sequence protocols with phase change behavior for detecting *global triggers* in a decentralized fashion. We address two types of global predicate triggers in this section:

1. **Threshold Trigger**: Detect when a global average crosses a threshold value.
2. **Interval Trigger**: Detect when a global average crosses outside a pre-specified interval range.

Concretely, suppose each process p proposes or maintains a value R_p for a parameter of interest R. We assume R takes values only in a finite range. The *average threshold trigger* detects the situation whether $R_{avg} = \sum_p R_p/N$ is above or below a pre-specified threshold value. The *average interval trigger* detects whether R_{avg} is inside or outside a pre-specified constant interval. Existing solutions for such global predicate triggers require a continuous aggregate calculation algorithm, e.g., [3], making them too expensive.

Instead, such a trigger (threshold or interval) can be implemented more efficiently through any sequence protocol that exhibits bifurcation (i.e., for a given constant r, respectively at a single value or at end-points of an interval). Each process runs the sequence protocol with a value of r_p derived from R_p through a globally consistent function that linearly maps the range of R into the range of r in such a way that the required threshold (resp. interval) maps onto the bifurcation point (resp. two bifurcation points).

For instance, partitioning of a peer-to-peer overlay can be detected by triggering an alert if the average membership list size falls below a threshold. Thus R_p =size

of membership list size at process p. Suppose the range of R_p is $[0, 10000]$, and we want to detect when R_{avg} crosses below a value of 100. As we show below, the Multiplicative protocol can be used to achieve this, with each process using $r_p = R_p/100$.

A phase change has either *high sensitivity* or *low sensitivity*, depending on whether the system behavior changes dramatically different or not, at the bifurcation point. For example, a bifurcation where a stable point becomes unstable is highly sensitive, while one where a stable point splits into two stable points has low sensitivity. The Multiplicative protocol has high sensitivity, while the Logistic protocol has low sensitivity.

3.1 The Multiplicative Protocol

The Multiplicative protocol is derived from the multiplicative equation $x_{m+1} = r.x_m$ ($r > 0$), and is shown in Figure 13 of Appendix A. Below, we first detail how threshold detection works with this protocol. Then, we present simulation results from an implementation.

3.1.1 Protocol Analysis and Threshold Detection

Equilibria and Phase Changes: Equilibrium occurs when $x = r.x$, or $x.(1 - r) = 0$. When $r \neq 1$, the only equilibrium point is $x = 0$. We ignore $r = 1$, since all values of x are equilibria. For stability of equilibria, we use a well-known result from Lorenz [12]:

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

For the multiplicative map, $f'(x) = r$ for all x. If $r < 1$, the system converges towards the sole stable point $x = 0$. If $r > 1$, this point turns unstable, and the system will diverge away from it and cap out at $x = 1$.

Convergence Times: For $r < 1$, with all N processes initially in state x, the expected number after t rounds is $(N.r^t)$. The expected convergence time is $log_{1/r}(N)$ rounds. Similarly, for $r > 1$, with a single initial process in state x, the number after t rounds is r^t, which gives an expected convergence time of $log_r(N)$.

Average Threshold Detection: Suppose each process uses a value of r_p for the Token generation actions, where r_p is derived linearly from R_p. It should be evident from the multiplicative map that replacing r with $r_{avg} = \sum_p(r_p)/N$ retains the same behavior. When $r_{avg} < 1.0$, the systems stabilizes with all processes out of state x; when $r_{avg} < 1.0$, the system stabilizes with all processes in state x. This is a sensitive bifurcation.

A small *control group* consisting of a limited subset of processes in the system (chosen randomly) is used to detect the trigger. At the start of each protocol period, the states of the processes in the control group is queried, and this is extrapolated to estimate the fraction of processes in the system that are in state x. If the fraction of control group processes that are in state x is close to 1.0 (resp. close to 0.0), then r_{avg} is above (resp. below) 1.0. If the fraction is close to neither 0.0 nor 1.0, then the protocol needs to be run for some more time.

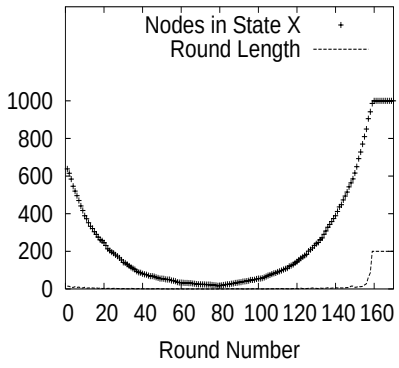

Figure 2: **Multiplicative Protocol: Basic Phase change.** The value of r_{avg} changes from 0.95 to 1.05 at t=80. $N = 1000$.

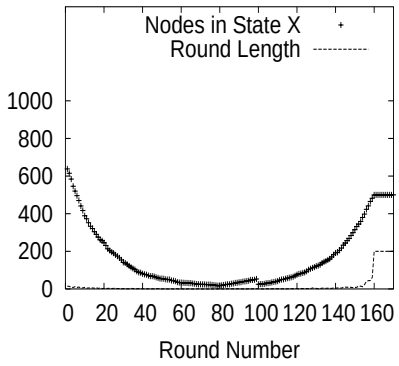

Figure 3: **Multiplicative Protocol: Massive Failure.** Same as Figure 2, except that 50% of the processes fail at t=100. Convergence occurs at t=206. $N = 1000$.

The Multiplicative protocol can be initiated for either *one-shot* detection or *continuous* detection. The one-shot mode is initiated by setting a small number of processes (not the control group) in state x. The continuous version periodically requires some processes (not the control group) to be forced into state x, since $r < 1$ could cause the fraction in state x to reach 0.0.

3.1.2 Experimental Results

The continuous Multiplicative protocol was implemented in C, and tested in a simulated environment, with multiple instances running synchronously (in periods) over a simulated network, all on a single machine (1.7 Ghz Intel Celeron CPU, 256 MB RAM, WinXP Pro). The Mersenne Twister pseudorandom generator is used. There are $N = 1000$ non-faulty processes. Round length is limited to 200 periods (each process receives and sends tokens once per period); plots show the actual round length.

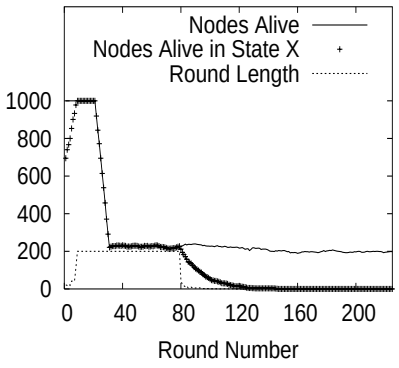

Figure 4: **Multiplicative Protocol: Churned System.** System begins to churn at t=20. The value of r_{avg} changes from 1.05 to 0.95 at t=80. Convergence occurs at t=147. Total of $N = 1000$ are churned.

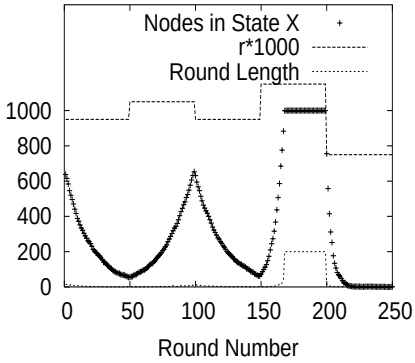

Figure 5: **Multiplicative Protocol: Varying** $r = r_{avg}$**.** $N = 1000$**.**

Figure 2 shows the basic effect of changing r_{avg}. Each process p sets r_p to be in an interval of size 1 around the chosen r_{avg}. r_{avg} is 0.95 from t=0 to t=80, and is 1.05 from t=80 onwards. The convergence towards $x = 0$ and $x = 1.0$ respectively in these two regions is evident. Notice also that when $r_{avg} > 1.0$, the actual round length reaches the maximum 200 periods, thus not all tokens are consumed. However, this does not not affect convergence.

Figures 3 and Figure 4 show the effect of massive failure and churn respectively. Notice that the number of nodes in the churned system is about 200. Failure of 50% of the processes delays convergence time only slightly, while churn has no perceivable change in the behavior (actually speeds up convergence due to the smaller system size). Finally, Figure 5 shows that the Multiplicative protocol is very sensitive to r_{avg} crossing the threshold.

3.2 The Logistic Protocol

The Logistic protocol is derived from the logistic sequence equation:

$$x_{m+1} = r.x_m.(1 - x_m), \text{With } r \geq 0, 0 \leq x_i \leq 1 \quad (3)$$

This equation was first used by the Belgian sociologist and mathematician Pierre Francois Verhulst (1804-1849) to model the growth of populations with limited resources. The Logistic protocol is derived from this equation by using the translation technique of Section 2 for polynomials terms. Informally, in this protocol, each process p checks at the beginning of a round whether it is in state x. 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 x, does process p generate an average r tokens for that round. These tokens are relayed and applied in the usual manner.

3.2.1 Protocol Analysis and Interval Detection

To find the equilibrium points of the protocol, set $x = r.x.(1 - x)$. This gives us either $x = 0$ or $x = 1 - 1/r$. Notice that the origin is an equilibrium for all values of r, while $x = 1 - 1/r$ is an equilibrium only if $r \geq 1$.

To determine the stability of these points, we apply Lorenz's condition. Since $(r.x.(1 - x))' = r.(1 - 2x)$, 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 Detection: As r is increased from 0, the only stable point is $x = 0$. As r crosses 1.0, the origin loses its stability; instead, a new stable point is created at $x = (1 - 1/r)$. This is a sensitive bifurcation, and can be used to detect when the value of r used in the system crosses a lower threshold value of 1.0. 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 [12]. 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 $f(q) = p$. This is called a flip bifurcation [2], and it is *insensitive*.

Thus, the Logistic protocol can be used to detect a global interval trigger by mapping the range of the parameter of interest R into the range of $r \in [0, 3.5]$ in a piecewise linear manner. We do not consider $r > 3.5$ due to possible chaotic behavior.

Convergence Times: We analyze the convergence times only for $0 < r < 3$, ignoring $r > 3$ because of the 2-cycle nature of that range. For $r < 1$, with N nodes initially in state x, the expected number after t rounds is upper-bounded by $N.r^t$. The expected convergence time is $log_{1/r}(N)$ rounds.

For $r \in [1, 3]$, some values of r have closed form solutions [14, 15]. For example, at $r = 2$, the solution is

[2] In fact, 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.

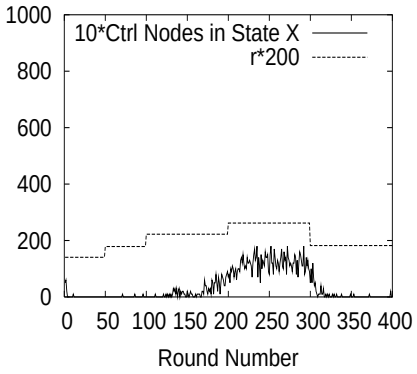

Figure 6: **Logistic Protocol: Lower Interval boundary of $r_{avg} = 1.0$.** As r_{avg} crosses above a value of 1.0, the system has a non-zero stable number of nodes in state x, while below 1.0, this number quickly drops to zero. $N = 1000$.

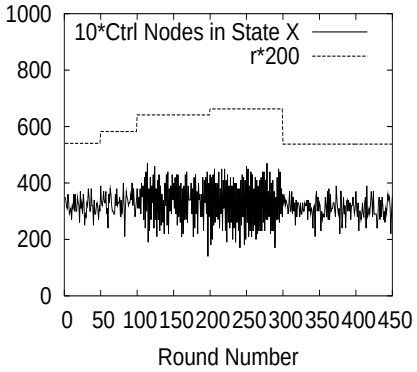

Figure 7: **Logistic Protocol: Higher Interval boundary of $r_{avg} = 3.0$.** As r_{avg} stays below a value of 3.0, the system is comparatively more stable than when > 3.0. $N = 1000$.

$x_m = \frac{1-e^{-2^n.c}}{2}$, where c is some constant [14], giving a convergence time of $O(log(log(N))$. Other r values may take longer to converge, but this is always upper-bounded by $O(N)$ since x_m is monotonic with m.

The Logistic protocol can also be executed in one-shot or continuous mode. The latter variant requires periodic forcing of a few processes into state x, and a few others out of state x (neither overlapping with control group).

3.2.2 Experimental Results

Th experimental conditions for the continuous Logistic protocol are the same as in Section 3.1.2. A small control group of 50 processes is used. Figure 6 and Figure 7 show the respective behavior when r_{avg} is near the lower and upper boundaries of the interval $[1.0, 3.0]$. Each process p sets r_p to be in an interval of size 1 around r_{avg}.

While the behavior at the lower interval is prominent (sensitive) as the stable number of processes in state x goes between being zero and being a positive quantity, the behavior change at the upper interval can be identified only by the increased perturbation (see Figure 7). This is due to the low sensitivity of this bifurcation. The protocol is resistant to failure and churn; we exclude plots due to space limits.

4 HoneyAdapt: A New Paradigm for Adaptive Grid Computing

Consider a Grid Computing problem that involves processing a large input data set. We assume a *master host* that hands out data chunks to other *clients* (workers), connected amongst each other through an overlay. All chunks are assumed to be the same size. Each client gets one chunk at a time. The clients may use 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 given data set. This is especially true of data sets with mixed chunks, where individual chunks may have a different "best" algorithms.

In this section, we first present a novel *adaptive* Grid Computing paradigm called *HoneyAdapt*. HoneyAdapt allows clients to adapt among A component algorithms *on the fly*, i.e., at run time. HoneyAdapt is derived from sequence equations that represent the behavior of honeybees as they attempt to select, in a decentralized manner, the "best" nectar source(s) from among multiple available sources [4, 11].

When applied to the sorting problem (Section 4.3), *HoneySort outperforms both component sorting algorithms - Quicksort and Insertion sort.*

4.1 The HoneyAdapt Sequence Protocol

We describe the sequence protocol below (derived using techniques for fractional, polynomial, and recursive translation). For clarity, the initial description also refers to the sequence equation and the honeybee behavior. Consider a group of clients (honeybees) trying to select the best from among A multiple algorithms (nectar sources) based on individual running times for the chunks at the clients (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 *forages* 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 [4]:

$$\begin{aligned} a_i\,(t) &= a_i\,(t-1)\,.\,(1-pf_i) \\ &+ \left(\frac{sq_i.pd_j.a_i\,(t-1)}{\sum_{j=1}^{A} sq_j.pd_j a_j\,(t-1)}\right).\sum_{j=1}^{A} pf_j.a_j\,(t-1) \end{aligned} \quad (4)$$

Here, $\sum_{i=1}^{A} a_i(t) = 1$ and all $a_i(0)$'s are non-zero. For each i, sq_i is the *quality* of algorithm i (nectar source i). 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 on a data chunk, it calculates a quality for i that is inversely related to the running time for that

chunk. sq_i is thus a quantity that is calculated on the fly, and depends on the nature of the chunk and i.

During each round, each client processes only one chunk. The algorithm choice is based on *advertisements* from previous rounds; the choice is random for the first round. Suppose client p uses algorithm $i(\in \{1, 2, \ldots A\})$ during a round. p calculates the "quality" $sq_{i,p}$ of i, based on the execution time in that round; a shorter execution time translates to a higher quality. sq_i in equation (4) 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. Dancing means generation of a number of *advertisement messages supporting algorithm* i. The number of advertisements generated is proportional to quality $sq_{i,p}$ in the round. These messages are then multicasted to *all* other clients.

Then, with following probability pf_i, 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. If p does not to follow (with prob. $1 - pf_i$), it continues to use old algorithm i.

4.2 Analysis of the HoneyAdapt Protocol

Lemma 4: *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., when:

either $a_i(t-1) = a_i(t) = 0$ or $\frac{pf}{sq_i pd} = \frac{\sum_j pf a_j(t)}{\sum_j sq_j pd a_j(t)}$

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 this implies $sq_i = sq_j$, a contradiction. Hence proved. □

Lemma 5: *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: here, $a_{i_{max}} = 1$, where $\forall j \neq i_{max} : sq_{i_{max}} > sq_j$.*

Proof: For this proof, Appendix B first extends Lorenz's condition to a multi-variable sequence equation. 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:

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

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

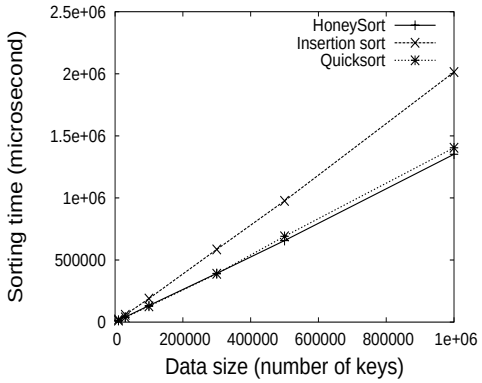

Figure 8: **HoneySort on random input.** With chunk size approx. 1000 and with 6 client machines.

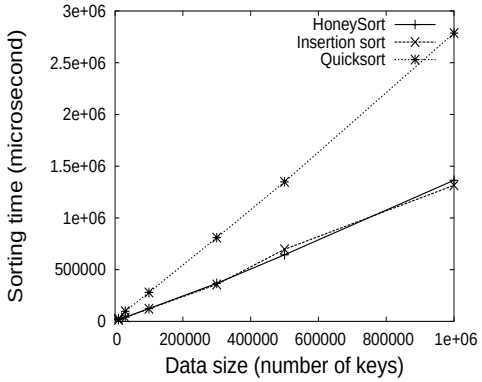

Figure 9: **HoneySort on pre-sorted input.** With chunk size approx. 1000 and with 6 client machines.

Convergence Times: Consider a small non-null perturbation from the equilibrium point $a_A = 1$. From Lemma 5, 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. Notice that the message traffic depends on parameter pd, but pd does not affect either equilibria or convergence speed.

4.3 The HoneySort Protocol, and Experiments

HoneySort is a variant of HoneyAdapt for the distributed sorting problem, with component algorithms of Quicksort and Insertion sort - other components are possible. Initially, the master partitions the input array into approximately equal sized chunks by using multiple pivots. At the end, the master simply concatenates sorted chunks.

HoneySort (Figure 14 in Appendix C) relaxes many of HoneyAdapt's assumptions. HoneySort is *completely asynchronous and does not use the concept of rounds at all*. Thus, all synchronization requirements are relaxed. When a client finishes sorting a chunk, it fetches the next chunk immediately from the master host. In addition to the normal following with pf, HoneySort picks a random algorithm with *randomization* probability pr; this maintains algorithm diversity. When a chunk is done, a *single*

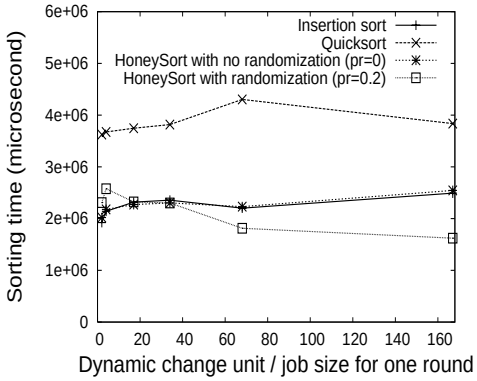

Figure 10: **HoneySort with randomization on dynamic input data.** The randomization probability pr is shown. With 6 client machines. Input array has 1 million entries, chunk size approx. 3000 elements, and a total of 333 chunks.

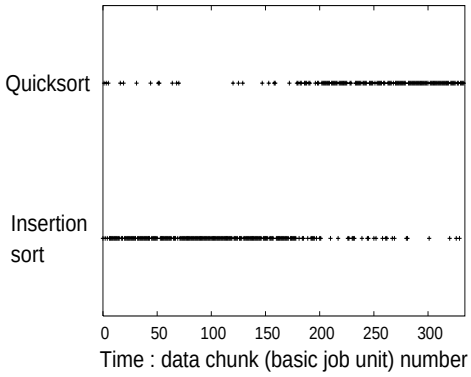

Figure 11: **Internals of HoneySort adaptivity.** Randomization probability $pr = 0.2$. Former half of input data is pre-sorted, the latter half random. With 6 clients. Input array has 1 million entries, chunk size approx. 3000 elements, and a total of 333 chunks.

advertisement message (with TTL=1) with the chunk's sorting quality sq_i is sent into overlay (eliminating extra messages). Advertisement sampling is modified accordingly. The master host has no part in advertisements.

We have deployed HoneySort on a Linux PC cluster. A TCP-based overlay is used (by default a complete graph). The *SortGen* microbenchmark [5] is used to create input arrays with 8 B integer elements. Figures 8 and 9 show running times respectively for completely random and pre-sorted input arrays. HoneySort is as fast as the best algorithm - Quicksort and Insertion sort respectively.

The next experiment involves dynamic arrays, which are partly random and partly sorted. Groups of *dynamic change unit* chunks within the input array are alternately random and sorted. If dynamic change unit=20, the first 20 chunks are sorted, the next 20 randomized, the next 20 sorted, and so on. Figure 10 shows that HoneySort outperforms both Insertion sort and Quicksort.

Figure 11 shows the internals of HoneySort's quick adaptation to data properties. Finally, Figure 12 shows that reducing the client degree in the overlay (neighbors chosen randomly) affects running times by under 35%.

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

Figure 12: **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.

This is partly because the degree affects only pd, which in turn does not affect protocol behavior.

5 Summary

We presented a new design methodology to translate multi-variable sequence equations into equivalent distributed protocols with the same trajectories and equilibria. Phase changes in the sequence protocols were used to efficiently detect global triggers such as average threshold (Multiplicative protocol) and average intervals (Logistic Protocol). Finally, a model of honeybee coordination was used to design HoneyAdapt, a new adaptive Grid computing paradigm. A variant called HoneySort is found to adapt on the fly to input data, outperforming distributed versions of both Quicksort and Insertion sort.

References

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

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

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

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

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

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

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

[8] G. S. Manku, "Balanced binary trees for ID management and load balance in distributed hash tables", *Proc. ACM PODC*, 2004, pp. 197-205.

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

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

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

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

[13] J. Wylie et al, "A protocol family approach to survivable storage infrastructures", *Proc. FUDICO*, 2004.

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

[15] Determinantal solution of the logistic map *http://www.iop.org/EJ/article/0305-4470/31/7/003/ja31007l3.html*

Appendix A: Multiplicative Protocol - Pseudocode

boolean x_{last};// state at start of this round
boolean x_{next}; // next state - running variable

Round m**:**
int $numtokens$;

set $x_{last} := x_{next}$;
set $x_{next} := 0$;

if (x_{last}==1) // am in state x at the start of this round
Generate $numtokens = \lfloor r \rfloor$ tokens;
With probability $(r - (\lfloor r \rfloor))$, generate an additional
token and $numtokens++$;
if (total number of tokens generated > 1)
set $x_{next} := 1$;
$numtokens := numtokens - 1$;
else
set $x_{next} := 0$;
if ($numtokens > 0$)
select $numtokens$ distinct non-faulty target processes,
each chosen uniformly at random from the group
(retry if targer unresponsive);
send each target process a message containing one
positive token;

//Token Relay and Apply
while (round is not over)
if (receive message with a positive token)
if(x_{next}==1);
select one distinct non-faulty target process,
chosen uniformly at random from the group
(retry if target unresponsive);
send to the target process a message containing
one positive token;
else
set $x_{next} := 1$; // consume token

Figure 13: **The Multiplicative Sequence Protocol.**

Appendix B: Lorenz's Condition extended to Multi-Variable Sequence Equations

Lemma A.1*(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 $\vec{\delta}$.

$$\begin{pmatrix} x_1(t+1) \\ x_2(t+1) \\ \vdots \\ x_m(t+1) \end{pmatrix} = \begin{pmatrix} H_1\left(\vec{x}(\infty) + \vec{\delta}\right) \\ H_2\left(\vec{x}(\infty) + \vec{\delta}\right) \\ \vdots \\ H_n\left(\vec{x}(\infty) + \vec{\delta}\right) \end{pmatrix}$$
$$\simeq \begin{pmatrix} H_1(\vec{x}(\infty)) + \frac{\partial H_1}{\partial x_1}\delta_1 + \ldots + \frac{\partial H_1}{\partial x_m}\delta_n \\ \vdots \\ H_n(\vec{x}(\infty)) + \frac{\partial H_n}{\partial x_1}\delta_1 + \ldots + \frac{\partial H_n}{\partial x_m}\delta_n \end{pmatrix}$$
$$= \begin{pmatrix} \frac{\partial H_1}{\partial x_1} & \frac{\partial H_1}{\partial x_2} & \cdots \\ \frac{\partial H_2}{\partial x_1} & \frac{\partial H_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} + \begin{pmatrix} x_1(\infty) \\ \vdots \\ x_m(\infty) \end{pmatrix}$$
$$\Leftrightarrow \vec{x}(t+1) - \vec{x}(\infty) = \begin{pmatrix} \frac{\partial H_1}{\partial x_1} & \frac{\partial H_1}{\partial x_2} & \cdots \\ \frac{\partial H_2}{\partial x_1} & \frac{\partial H_2}{\partial x_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} \tag{5}$$

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 $\vec{\delta} = c_1.V_1 + c_2.V_2 + \ldots + c_m.V_m$, 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. □

Appendix C: HoneySort Protocol - Pseudocode

```
// The process in the master machine
Master() :
    sortingkey data[], *(chunks[]);

    Establish a data transfer overlay to all clients

    //To ensure no further processing is needed
    // after each chunk is sorted by clients,
    // partition the data into chunks of nearly equal size,
    // where data is left unordered within a chunk, but ordered across chunks,
    // using Sample sort technique.
    chunks[] := pivot_sample(data);

    // Receive sorted data and send new data on-demand
    while (not all chunks[] are done)
        if (receive sorted chunk from client i)
            send next chunk to client i;

// The process in a client machine
Client() :
    sortingkey chunk[];
    int sorting_algorithm;
    double pd, pf, pr;
    message advertisement;

    Connect to master for data transfer and instruction
    Join client overlay

    // Initialize values according to the master's instruction
    set sorting_algorithm := the initial sorting algorithm;
    set pd := probability to dance/advertise algorithm;
    set pf := probability to follow/choose other algorithm;
    set pr := probability to randomly select algorithm;

    while (chunk[] = receive_data())
        sort chunk[] with sorting_algorithm;
        send back chunk[] to master;
        if (with probability pd)
            set advertisement := {sorting_algorithm, its speed};
            send advertisement to client overlay neighbors;
        if (with probability pr)
            set sorting_algorithm := randomly picked algorithm;
        else
            if (with probability pf)
                gather all advertisement from peers;
                for each sorting algorithm, evaluate its score as sum of all its speed;
                set sorting_algorithm :=
                    pick algorithm with probability proportional to its score;
            else
                ; // use the same sorting_algorithm for the next round
```

Figure 14: **The HoneySort Protocol.**