Book Title
Book Editors
IOS Press, 2003

Systematic Design of P2P Technologies for Distributed Systems

Indranil Gupta
Dept. of Computer Science
University of Illinois at Urbana-Champaign, Urbana IL 61801
Ph: 1-217-265-5517. Fax: 1-217-265-6494.
indy@cs.uiuc.edu

Abstract. While several peer-to-peer (p2p) schemes have emerged for distributed data storage, there has also been a growing undercurrent towards the invention of *design methodologies* underlying these peer-to-peer systems. A design methodology is a *systematic technique* that helps us to not only design and create new p2p systems (e.g., for data storage) in a quick and predictable manner, but also increases our understanding of existing systems. This chapter brings together in one place previous and existing work by several authors on design methodologies that are intended to augment the activity of p2p algorithms, keeping our focus centered around (but not restricted to) data storage systems. As design methodologies grow in number and in power, researchers are increasingly likely to rely on them to design new p2p systems.

Keywords. Distributed Protocols, Design, Methodologies, Declarative Programming, Differential Equations, Composable Methodologies.

1. Introduction

Today, researchers design large-scale distributed systems (such as p2p data stores) mostly by using an ad-hoc approach, with literature, experience, and education as the only available aids. This has arguably resulted in very complex distributed systems [20] and great difficulty in understanding the properties of these systems. Worse still, this has caused increased unreliability in today's systems [6], and a phenomenon that Tanenbaum calls "software bloat" [22].

However, in the past few years, a new series of techniques has emerged to both (1) simplify our understanding of existing distributed systems, as well as to (2) help us to design new distributed systems in a systematic manner. We refer to these two uses above as [12]: (1) retroactive and (2) progressive uses respectively.

A design methodology for a distributed protocol (or distributed system) can be loosely characterized as [12] "an organized, documented set of building blocks, rules and/or guidelines for design of a class of distributed protocols. It is possibly amenable to automated code generation."

To start off, let us briefly look at two examples of very power design methodologies.

For instance, Loo et al recently designed a system called P2 [15]. P2 is generally speaking a declarative language that can be used to specify the topologies of p2p overlays such as Chord and multicast overlays such as Narada. In effect, the researcher designing a system can not only formally specify the rules for the overlay topology, but also potentially formally (and automatically) verify properties of the system, as well as generate code for the system! One argument goes that had Chord been originally designed using P2, the entire system would have taken but half a day to design and code up, not the several months that it took its human designers! Further, as the authors of P2 showed, this automatically designed system would have performance that is quite close to that of the hand-designed Chord.

The second methodology we wish to point to is one that translates certain classes of differential equations into *equivalent* distributed protocols [11]. The derived protocols are replicated state machines with local and simple actions and transition rules, but most importantly, the globally emergent behavior of the protocol is equivalent to the behavior of the original differential equations. Differential equations have been used by several scientists (especially non-Computer Scientists) to represent ideas and results - this design methodology now allows protocols to be systematically derived from these natural models *predictably and without any side effects.* Article [11] then goes on to show how endemic disease models can be used to build distributed storage systems, and how other models (all represented as differential equations), e.g., ecological models, can be translated into protocols for important distributed computing problems.

The above methodologies are only indicative of the power of design methodologies. The rest of this chapter will expand on the details of some of these methodologies, and the reasons why they are good starting grounds to invent both new methodologies as well as new distributed systems.

In Section 2, we present a taxonomy of design methodologies, not necessarily restricted to distributed systems only. Section 3 discusses declarative programming models for distributed system design, and Section 4 describes the translation of differential equations into distributed protocols. Section 5 briefly outlines several other emergent and mature methodologies for p2p systems. We summarize the chapter in Section 6.

A note is due on the material in this chapter - all the material covered in this chpater has already been presented by researchers previously, hence in a sense, this chapter is akin to a survey paper on this new topic, albeit with hints at future directions. Thus, it is unavoidable and inevitable that this chapter reuses and in some cases reproduces, certain defintions, tables, descriptions of algorithms from the original publication. This is done mainly to ensure that the algorithms and designs proposed by the original authors of the respective paper(s) are presented faithfully. Wherever such reproduction occurs, we reference the original text, even if it is a reference to one of our own prior papers. Nevertheless, the reader will find that this reproduction is not a mere cut-and-paste: the material has been sufficiently edited to make it fit into the flow of the chapter and tie it to additional discussion. It is hoped that this approach of ours will preserve both the context and yet present the referenced authors' work in the best light.

METHODOLOGY TYPES	**Innovative**	**Composable**
Formal	Protocols from Differential Equations [11]; Bluespec for hardware synthesis [1]; P2 language for P2P systems [15]; RAML language for routing design [9].	TCP/IP layered architecture; Extensible router and OS designs (e.g., Click [14], SPIN, x-kernel [4,25]); Routing [28]; Probabilistic I/O automata [26]; Strategy Design Patterns [8]; Stacked architectures (e.g., Horus [24]).
Informal	Design Patterns [7].	DHT design methodologies [13,18]; Protocol family for survivable storage [27]; Probabilistic protocols [23].

Table 1. How Existing Methodologies Fit into the Proposed Taxonomy (part of table borrowed from [12]).

2. Taxonomy of Methodologies

In order to motivate an understanding of the features of methodologies, in this section, we first discuss a taxonomy of classification for them. This taxonomy first appeared in article [12]. In order to define our taxonomy, we have to define several terms - our definitions for these terms thus require unavoidable borrowing from the article [12].

Formal vs. Informal: A methodology that is specified using precise rules or a stringent framework is called a *formal* methodology. Otherwise the methodology is said to be *informal*. These "rules" for a formal methodology could either be mathematical or logical notation, or the grammar of a high level programming language. Respective examples are the probabilistic I/O automata [26], and the methodology of [11] that takes as input a set of differential equations (satisfying certain conditions), and generates code for a distributed protocol that is equivalent. Previous methodologies for DHT design [13,18] have been informal. We will discuss all these methodologies at different points during this chapter.

Due to their rigor (either through a formal framework or a compiler), formal methodologies can be used to create protocols with predictable or provable properties, and also to generate protocol code automatically. For example, the distributed protocols generated from differential equations in [11] are provably equivalent to the original differential equation, and can be generated by a toolkit called DiffGen [11]. On the other hand, informal methodologies are less rigorous and more flexible, but can have multiple possible interpretations. An informal methodology could be converted into a formal one through implementation of a specific interpretation. For example, an informal probabilistic protocol composable methodology can be instantiated through a high level language called Proactive Protocol Composition Language (PPCL) [10,23], thus making it formal. Once again, we will discuss each of these methodologies at different points during this chapter.

Innovative vs. Composable: Design methodologies must be capable of assisting in innovation of new protocols, as well as in the ability to reuse and adapt existing protocols. These are achieved respectively through innovative methodologies and composable methodologies. An innovative methodology describes how completely novel protocols can be created, e.g., [7,11]. A composable methodology typically describes *building blocks* and *composition rules or guidelines*. Building blocks are either standalone protocols or strategies, and composition rules help combine the blocks to create new protocols with enhanced properties. For example, the informal methodology for DHT design in [13] uses four types of building blocks - overlay, membership, routing, and preprocessing. Strategy design patterns are another example of a composable methodology [8].

Table 1 summarizes the above discussion.

Discovery of Methodologies: Different approaches are possible for the *discovery* of these methodologies:

1. **Retroactive**: A methodology is discovered for an existing system or class of protocols. Ex: methodologies for routing [28] and probabilistic protocols [10].

2. **Progressive**: A methodology is invented that creates a novel class of protocols. Ex: the design of protocols from differential equations can generate new protocols for dynamic replication and majority voting [11].

3. **Auxiliary**: A methodology is discovered to assist and complement an existing methodology. Ex: protocol families for survivable storage architectures [27] combine several auxiliary methodologies for differing system models.

3. Declarative Languages for Protocol Design

Recently, there has been emergence of several declarative paradigms for specifying the design of distributed systems. As opposed to imperative programming languages (such as C or Java) that explicitly describe the detailed actions of a distributed protocol (i.e., the *how*), a declarative approach to programming instead specifies the low-level *goals* of the distributed protocol (i.e., the *what*), leaving the translation of this specification into actual code to a separate compiler or interpreter (which in turn can perhaps be changed by the user).

Declarative programming can be achieved by either a functional programming language (ML-like) or by a logic programming language (Prolog-like) or a constraint language. Below, we first describe P2 a declarative logic programming that can be used to design peer to peer systems (and eventually in distributed data stores). Then, we briefly describe RAML, a language that can be used to design routing protocols. Although the second case study is not directly related to storage, it is related to routing, an important component in any distributed store.

Notice that both the methodologies presented below are formal and innovative methodologies, although the second methodology (Section 3.2) is also amenable to composition. Both these methodologies have both retroactive and progressive uses.

3.1. P2: Declarative Design of P2P Overlays

B.T. Loo et al's P2 is a declarative logic language intended for the design of overlays as well as multicast protocols [15]. Since overlays are directly applicable in designing distributed stores, below, we describe the application of P2 to design of a simple, canonical overlay. P2 is based on Datalog, a general declarative query language that is a "pure" subset of Prolog that is free of imperative constructs.

Consider a peer to peer system that is structured as a logical ring - each node lies at some point in this ring and has a successor and a predecessor. This ring overlay is the most canonical form of the Chord peer to peer system [21], and we will describe the use of P2 in designing the ring overlay. Article [15] details the design of the entire Chord system (which has several non-successor neighbors for each node in order to speed up search).

P2 uses several rules to specify what the system needs to achieve. Each of these rules applies at each node that is participating in the overlay. Below, we describe some of the rules:

•$materialize(succ, 120, infinity, keys(2))$

This rule specifies that each node in the system will maintain a table called *succ* (successor) whose tuples will be retained for 120 seconds and have unbounded size, while $keys()$ specifies the position of tuple field that is the primary key of the table.

•$stabilize(X) : -periodic(X, E, 3)$

Here, canonically, *stabilize* is a table with row that has value (X) for an X, if table *periodic* has a row with value $(X, E, 3)$ for some E. Specifically in P2 though, *periodic* is not a table – instead it is a built-in *stream* that periodically produces a tuple with a unique identifier E at node X (in the distributed system), in this case once every 3 seconds. This also means that *stabilize* itself is not a table maintained at a node, but instead an event generated at the node. As we will see below, this *stabilize* event further causes the node to refresh the successor tables of its neighbors, as well as to increment its own successor table version number.

•$lookupResults@R(R, K, S, SI, E) : -node@NI(NI, N),$
$lookup@NI(NI, K, R, E), succ@NI(NI, S, SI), Kin(N, S]$

Each object in this peer to peer system has a unique identifier that lies somewhere along the ring overlay. When a query or an insert request is generated by some node in the system for a given object (with the object id), it needs to be *routed* to the appropriate node in the ring overlay, i.e., the node that lies right after the object id's location in the logical ring.

The above described rule returns a succesful lookup result if the received lookup seeks a key K found between the receiving node's identifier and that of its successor.

•$sendSuccessor@SI(SI, NI) : -stabilize@NI(NI, \_), succ@NI(NI, \_, SI)$
•$succ@PI(PI, S, SI) : -sendSuccessors@NI(NI, PI), succ@NI(NI, S, SI)$

Finally, the above two rules come into play when the node (periodically) needs to refresh its successor table (and that of its table). In the first rule above, a node asks its successors (all if there are multiple successors) to send it their own successors, whenever the *stabilize* event is issued at that node. The second rule above then installs the returned successor at the original node.

In summary, we have taken only five rules above to specify the design of the entire ring overlay - from basic tables to lookup to stabilization. Article [15] describes the design of the entire Chord protocol (substantially more complex than the above ring overlay) in merely 47 rules!

Some advantages of such a declarative approach to design are evident; other benefits are not so obvious:

1. Ease of Protocol Specification: A protocol designer no longer has to write a C/C++/Java program several thousand lines long to design a new system. Design is a matter of writing only a few rules.
2. Formal Verification: Any such declarative design can potentially be run by specially-built verification engines that find bugs in the design, or better still, analyze the scalability and fault-tolerance of the protocol.
3. On-line distributed debugging: Execution history can be exported as a set of relational tables, distributed debugging of a deployed distributed system can be achieved by writing the appropriate P2 rules.
4. Breadth: The same language P2 can be used to design other p2p overlays beyond Chord (e.g., the Narada overlay) - this makes possible quantitative comparisons among these systems that are much more believable than mere simulation-based comparisons. In addition, hybrid designs can be explored.

Clearly, the downside to this approach is the learning curve associated with "yet another new language". Yet, if the results of article [15] are to be believed, the performance obtained by a system designed using P2 is comparable to the original hand-coded system. In view of the above benefits, many researchers might consider it worth to learn another programming language, especially if it makes the difficult job of system design a little bit easier.

3.2. RAML: Declarative Design of Routing Protocols

Routing protocols for the Internet are difficult to design and much more difficult to verify for correctness. There has been a lack of design methodologies for routing protocol design, leading to the over-use of fairly well-understood protocols, e.g., BGP is not intended as an IGP, yet is being pressed into service as IGP in several parts of the Internet - this is a problem since BGP convergence properties are not well-known. A different IGP protocol would need to be designed.

In [9], Griffin and Sobrinho present a new programming language called the *Routing Algebra Metalanguage (RAML)* which enables protocol designers to specify classes of *routing algebras* and manipulate the algebras, as well as verify several convergence and correctness properties about such algebras.

Below, we briefly give an overview of the Metarouting framework of [9], mentioning the authors' findings. For more details, the reader should refer to [9].

RAML Overview: The design of any routing protocol consists of both policies and mechanisms. Policy defines how the attributes of a route are described, what defines a route as a "best" route, etc. The mechanism defines how routing messages are exchanged, what route selection algorithms are exchanged etc. Clearly, in any routing protocol, there is an inter-play between the corresponding parts of policy and mechanism.

The RAML language is based on a metarouting framework that defines a tuple:

$RP =< A, M, \ldots >$,

where A is a routing algebra (the policy part) and M is a set of mechanisms that can be associated with a routing adjacency. Several mechanisms can be used for the same protocol. We will ignore the rest of the above tuple for the purposes of simplicity in this chapter. The algebra is represented as:

$A = (\Sigma, <=, L, \oplus, \ldots)$

In the above description of a routing algebra A, Σ is a path signature and $<=$ is a preference relation over Σ that is both complete (i.e., defined for all pairs of elements from Σ) and also transitive. Finally, L is a set of link labels and $\oplus$ is a relation from $L \times \Sigma \rightarrow \Sigma$.

This algebra now allows us to specify in place of $\oplus$ simple operators such as ADD, MULT, MAX, MIN, etc., so that we can now reason about paths. For instance, shortest path routing can be reasoned about by using $\oplus = ADD$ above and using the normal $<=$ relation to compare different paths in Σ.

More importantly though, for a given class of algebras, one can define important path properties. The main properties explored in [9] are: Monotonicity, Strict Monotonicity and Isotonicity - all these properties apply to pairs in $L \times L$ w.r.t. the operation $\oplus$. For instance, ADD satisfies all the above three properties.

The authors then go on to define operations among entire pairs of algebras, i.e., given two different routing algebras A_1 and A_2, one can define composite routing protocols based on, say, the lexical product of the two algebras. The authors also explore scoped product and disjunction-based composition. *Notice that this makes the RAML methodology in a sense both innovative and composable.*

Finally, the authors show how RAML can be used to define "better" IGP protocols than BGP - one description they present involves merely a 1-line specification (for A)! Yet, this algebra has provable convergence and correctness properties. Then, the authors go on to reason about BGP itself and specify it in their RAML framework. They show that the policy component of BGP is in fact a scoped product of the EBGP policy component's algebra and the IBGP's policy component's algebra. Here both EBGP and IBGP can each be represented as a RAML algebra.

This is an extremely powerful methodology - not merely for design (one can specify entire routing policies within just 1 line), but also for reasoning about the convergence and correctness properties of these policies. It is likely that RAML or RAML-like languages will be increasingly used by the networking community both retroactively as well as progressively, over the coming few years.

4. Methodologies for Nature-Based Design of Distributed Systems

Several fields of science including Biology, Chemistry, Physics, and several non-engineering fields such as sociology, etc. use differential equations to represent ideas and results. Translating ideas and results from these fields into distributed protocols in a systematic manner requires a design methodology. Several nature-inspired distributed systems have been designed, e.g., [2]. However, without a *systematic* design methodology being used for the translation, there is the risk that a protocol derived from a natural phenomenon may not be an accurate translation, or worse still have side-effects in distributed computer systems that are otherwise not seen in nature.

Here, we briefly present previously published results for an innovative methodology that translates sets of differential equations into *equivalent* distributed protocols - some of the discussion is inevitably borrowed from the article [11]. An equation set generates a state machine, with each variable mapped to a state, and terms mapped to protocol actions. Stable equilibrium points in the original equations map to self-stabilizing behavior (fractions of nodes in respective states), and simplicity of terms maps to scalable communication.

In brief, consider a system of differential equations in the form $\dot{\bar{X}} = \frac{d\bar{X}}{dt} = \bar{f}(\bar{X})$, where $\bar{X}$ is a vector of variables, f is a vector of $|X|$ functions, and the left hand sides denote each of the variables differentiated with respect to time t. An example with $X = \{x, y, z\}$ is:

$$\begin{aligned} \dot{x} &= -\beta xy + \alpha z \\ \dot{y} &= \beta xy - \gamma y \\ \dot{z} &= \gamma y - \alpha z \end{aligned} \qquad (1)$$

Here, α, β, γ are parameters lying in the interval $[0, 1]$. When the methodology of [11] is applied to the above equation, the protocol generated is *equivalent* to the original equation system, viz., *the fractions of processes in different states in the distributed system, at equilibrium, is the same as the values of the variables when the original equations are in equilibrium.*

Below, we describe in brief the protocol derived from the above differential equation system, as well as the general methodology. For more details on the details of the translation methodology, the reader is encouraged to refer to the original paper [11].

Endemic Protocol: Equations (1) above in fact represent the survival of endemic diseases (such as the flu) in a fixed-size human population, with x, y, z respectively the fractions of receptives, infected, and immune individuals. The protocol derived from it is a new *dynamic and migratory replication model.* In this dynamic and migratory replication scheme, once a given file is inserted into a distributed group of computer hosts connected in an overlay, the emergent behavior of the protocol ensures that the file has a small number of replicas moving continuously among the hosts in the distributed system (only "infected" hosts store a replica). Without using too much network bandwidth, this scheme ensures *attacker-resilience* – an attacker cannot accurately guess the current set of replica nodes for a given file.

Effectively, the protocol derived from equations (1) above can be represented as a replicated state machine that runs independently at each process in the distributed system. This replicated state machine is represented below. Here, (1) an "infected" host is labelled as being in a "Stash" state since it is storing the file, and (2) an "immune" host is labelled as an "averse" host for historical reasons.

The state machine derived has three states - x (susceptible or *receptive*), y (infected or *stash*), z (immune or *averse*). *A process is responsible if and only if it is in the stash state.* The actions are as follows.

Assume that $\beta \in (0, 1]$. State actions can be defined to be executed periodically, as discussed earlier in the methodology. (i) (γy term) A process p in the stash state tosses a coin with a biased heads probability γ - if the coin falls heads, process p changes its state to averse. This transition is accompanied by a deletion of the object replica at p. (ii) (αz term) A process p in the averse state tosses a coin with heads probability α, and changes state to receptive if this coin falls heads up. (iii) (βxy term) A process p in the receptive state chooses 1 target uniformly at random, and if the target is in the stash state, and a locally flipped coin (with heads probability β) falls heads simultaneously, process p changes its state to stash (after an object transfer).

The analysis of this protocol reveals that this distributed system has two equilibria:

$$\begin{aligned} &(x_\infty, y_\infty, z_\infty) = (1, 0, 0), \text{ and} \\ &(x_\infty, y_\infty, z_\infty) = (\gamma/\beta, \frac{1-\gamma/\beta}{1+\gamma/\alpha}, \frac{1-\gamma/\beta}{1+\alpha/\gamma}) \end{aligned} \tag{2}$$

We call these respectively as the *first equilibrium point* and the *second equilibrium point*. The first equilibrium results in all copies of the object disappearing; all processes eventually turn receptive. The second equilibrium is more desirable, since it guarantees a stable number of stashers in the system.

Surprisingly, the analysis in [11] shows that *starting the distributed system initially at any point* <u>*other*</u> *than the line* $y = 0$ *(includes the first equilibrium point) causes it to eventually (and quickly) converge towards the second equilibrium.* This leads the protocol to have the following properties: (1) Self-Stabilization: the system always tends to converge back to a stable (small) number of stashers, (2) Untraceability: it is probabilistically difficult for an attacker to guess who all the stashers are (the window until this set changes is very short), (3) Fault-tolerance: due to the properties (1) and (2) above, massive failures and churn are tolerated by the protocol, (4) Scale: the above properties are guaranteed by basically using constant bandwidth at each host, independent of the system size.

This dynamic and migratory replication model is currently being used to design a new distributed file system called "Folklore" [19].

The Translation Methogology: In order to clearly classify the differential equation classes that can be converted into distributed protocols, we first describe a taxonomy of differential equation systems. We focus only on equation systems that have a single differential per component equation, and are of order and degree 1. Such an equation is canonically $\dot{\bar{X}} = \bar{f}(\bar{X})$. We also denote the equation for variable $x \in X$ as $f_x(\bar{X})$. Further, each $x \in X$ lies in $[0,1]$ with differentials at the interval end-points being either zero or pointing in towards the other direction.

We define a *term* as an elementary unit of f_x. In other words, we express each $f_x(\bar{X})$ as a sum of elementary terms, where each term could be either positive or negative. Our translation methodology works on equation systems that are both *restricted polynomial* and *completely partitionable.*

An equation system $\dot{\bar{X}} = \bar{f}(\bar{X})$ is *restricted polynomial* if (i) it has only polynomial terms, and (ii) for each $f_x(X)$, it is true that each negative term $-T = -c_T \Pi^{y \in X} y^{i_{y,f_x,T}}$ that occurs in it ($c_T \in [0,1]$) has $i_{x,f_x,T} >= 1$. Equation system (1) is an example of a restricted polynomial system.

An equation system $\dot{\bar{X}} = \bar{f}(\bar{X})$ is said to be *completely partitionable* if and only if (i) it is completethe sum of all right hand sides of all the equations sum to zero, and (ii) the multi-set consisting of all terms in $\bar{f}(\bar{X})$ can be written as a union of mutually disjoint subsets, where each subset contains exactly two terms, and the two terms in each subset add up to zero (i.e., one term is the negative of the other in the pair). For example, equation system (1) is completely partitionable.

Now we describe two translation techniques for certain term pairs in the differential equation system. Consider a polynomial term of the form $-T = -c.\Pi^{y \in X} y^{i_{y,f_x,T}}$, occurring in f_x, with the corresponding $+T$ term occurring in f_y. If $-T = -c.x$, it can be translated into a *Flipping* action, otherwise it is translated into a *One-time-sampling* action. Each of these actions are executed periodically, with the time to the next action determined by a memoryless distribution with an average of $\frac{1}{k_x}$ time unit, where k_x is the number of negative terms on the right hand side of f_x - each next action is selected at random from among the k_x possibilities. Needless to say, the definitions of these terms are also borrowed from [11].

- *Flipping:* A term of the type $-c.x$ ($c \in [0,1]$) occurring on the right hand side (henceforth "r.h.s.") of $\dot{x} = f_x(\bar{X})$ is mapped to a flipping action. A process in state x *locally* tosses ("flips") a biased coin that has heads probability c. Only if the coin turns up heads, does the process transition out of state x; the new state is y, where $\dot{y} = f_y(\bar{X})$ contains the corresponding $+c.x$ term, and $y \neq x$. Notice that for one execution of the flipping action at a given process, the probability of success, and thus state transition, is c.

- *One-time-sampling:* A term $-T$ of the type $-T = -c.x^{i_{x,f_x,T}}.\Pi^{y \in X - \{x\}} y^{i_{y,f_x,T}}$ ($c \in [0,1]$) that occurs on the r.h.s. of $\dot{x} = f_x(\bar{X})$ with $i_{x,f_x,T} >= 1$ is mapped into the following action. A process in state x samples $(i_{x,f_x,T} - 1 + \Sigma^{y \in X - \{x\}}(i_{y,f_x,T}))$ other processes, each such target process selected uniformly at random from across the group. In addition, the process also flips *locally* a biased coin that has heads probability c. Let the variables $y \in X$ be ordered lexicographically. Then the process makes a transition out of state x (and into the corresponding state with the $+T$ term) if and only if (i) each of the first $i_{x,f_x,T} - 1$ target choices happen to be in state x; and (ii) for each $j : 1 \leq j \leq$), we have $\Sigma^{y \in X - \{x\}} i_{y,f_x,T}$, the j^{th}

process sampled is in the same state as the j^{th} variable in $\Pi^{y \in X-\{x\}} y^{i_{y,f_x,T}}$ (when ordered lexicographically); and (iii) the flipped local coin falls heads [1]. Notice that for one execution of this one-time-sampling action at a given process, the probability of success, and thus state transition, is $c.x^{i_{x,f_x,T}-1}.\Pi^{y \in X-\{x\}} y^{i_{y,f_x,T}}$.

The article [11] then goes on to show that the above two translation techniques are sufficient to translate *any differential equation system that is both completely partitionable and restricted polynomial*. This class is not small at all - it contains a large number of very useful differential equations, including the endemic process shown earlier in this section. In addition, this methodology can also be used to design a new protocol for majority voting in large-scale distributed systems (derived from a model of biological competition among ecosystem species), and a protocol from adaptive Grid Computing (derived from coordination models for honeybees). The reader is encouraged to see details in [11].

This methodology can be automated to generate code at the protocol designer's desk. The differential equation translation methodology has been incorporated into a design toolkit called *DiffGen*, which enables a designer to input differential equations (in a Mathematica-like format), and outputs compilable and deployable C code for the equivalent protocol.

The methodology just presented is extremely powerful in designing new distributed storage systems. The Folklore file system [19] creates an untraceable data store by building on top of the core endemic protocol described above. Article [11] also describes how differential equations representing certain ecosystem behaviors (the Lotka-Volterra model of competition) can be translated into a distributed protocol for majority voting, with potential use in distributed digital archives that need to ensure that the "good" copies of articles survive in spite of bit-rot and malicious attacks [17].

Intuitively, the protocols generated by this methodology are similar to Complex Adaptive Systems, and can be considered as a type of "Emergent Thinker" [5]. The methodology avoids any of the side-effects of informal translation mechanisms. Besides it revealing that natural phenomena can indeed be translated systematically into distributed protocols, the more surprising aspect of this direction of research has been that the generated protocols in fact (1) solve useful and practical problems, and (2) have many (and more!) characteristics that distributed systems designers consider important for large-scale distributed systems.

5. Other Methodologies

In the previous sections in this chapter, we have focused on a select set of recent developments in the field of design methodologies for distributed systems and distributed stores. In this section, we focus on some not-so-recent and some classical design methodologies for distributed systems. This section will wrap up our discussion on the topic.

[1]The idea behind this condition is of course similar to that behind the well-known *Law of Mass Action*. Flipping is in fact a special case of One-time-sampling.

5.1. Other Theoretical Methodologies

Probabilistic I/O automata and Composition: Lynch and Tuttle [16] originally defined a model of an I/O automaton that contains states, as well as transition rules defined among state pairs. The transition rules are based on *actions* possible in those states, where each action could be either one of the following three types – input, output, or internal actions.

Wu et al then extended this I/O automaton and were able to define a probabilistic I/O automaton in [26]. In this variant, for each state and each action that the state can receive, the specification defines probabilities of transition to all possible next-states. In addition, a delay function is defined, determining how long the system will wait in this state before moving to the selected next-state.

Most importantly though, Wu et al go on to describe how entire automata can be composed with each other. Then, they detail how automaton properties such as equivalence and other properties such as full abstraction.

In a sense, the probabilistic I/O automaton model is an advanced version of the differential equation-based automata derived in Section 4. It should be possible to further develop the translation mechanisms described for differential equations in Section 4 so as to enable the design of more complex (and thus more powerful) probabilistic I/O automata.

Automatic Discovery of Mutual Exclusion Protocols: In any distributed storage system, the ability to access resources (e.g., files) in a mutually exclusive manner (i.e., at most one process accesses the file at any time, with deadlock-freedom and starvation-freedom) is very important. Bar-David and Taubenfeld have described a methodology in [3] which allows the protocol designer to specify (via a new high-level programming language) several parameters that they require from a mutual exclusion algorithm. With these parameters, their tool then goes off and explores all possible programs (in pseudo-code) that satisfy the constraints specified, yet are correct solutions to the mutual exclusion problem.

The constraints that can be specified by the designer include: (1) number of processes (2) number of lines of code (3) number, size and type of variables (4) type of conditions: simple or complex. Complex conditions are composed of two simple conditions (terms) related by and, or, xor. The design of their tool contains three basic components – an internal algorithm generator, an algorithm verifier, and an optimizer.

The authors explore the feasibility of their design for small-scale distributed systems (up to a few processes only), yet this yields interesting results. For instance, the authors found that with a constraint of two shared bits among processes and simple conditions, any mutual exclusion protocol has to have at least 7 entry commands and at least 1 exit command. The author's tool also managed to find a mutual exclusion protocol that was shorter than a classical mutual exclusion algorithm previously proposed by Dekker - the new algorithm has merely 6 entry and 1 exit commands while Dekker's algorithm had 9 entry and 2 exit commands.

The benefits of using similar tools lies not merely in being able to generate simpler programs that solve the given problem correctly (although this has clear benefits such as reduction of code complexity, ease of debugging and maintenance). The benefit lies more in being able to make the protocol designer's job easier without taking away any of the flexibility or stifling the creativity involved in the protocol design process.

5.2. Other Informal Methodologies

Informal Methodology for P2P systems: Iamnitchi and Foster presented an informal methodology for the design of p2p resource discovery schemes in [13]. This composable methodology requires the design of four components:

1. Membership Protocol: Specifies how hosts maintain their neighbors, in spite of node join, leave and failure.
2. Overlay Construction: Specifies how neighbors are selected from among all possible candidates. Usually based on a set of invariants and rules.
3. Request Processing: Routing, through the overlay, of lookup and insert and delete requests to the appropriate responsible node in the system.
4. Preprocessing: Includes prefetching and usage-based and other optimizations.

In addition, nodes in a p2p overlay for resource discovery typically have unique (logical) nodeID's assigned to them. Manku [18] explores several techniques for flexible nodeID assignment and maintenance in p2p systems. He uses balanced binary trees for nodeID management and assignment. This adds a fifth component to the above design informal methodology.

Adaptive Data Stores: There is a wide variety of system designs and protocols for survivable storage infrastructures. Each design point addresses a particular fault model (e.g., crash-stop or Byzantine, repairable or non-repairable, synchronous or asynchronous). Thus the fault model assumption is typically incorporated in at the protocol design time.

Wylie et al [27] describe work on a mechanism that allows a distributed store to address a wide variety of fault models by making the fault-model decision at run-time, rather than at design-time. This is achieved by developing a framework where multiple protocols, each addressing a separate fault model, can be incorporated. Then, at run-time, the appropriate fault-models can be selected, either explicitly, or dynamically and automatically by detecting the most prevalent types of faults (through a separate detector).

This is an informal composable methodology that helps design an adaptive distributed storage system; one that can tolerate a wide variety of faults. This approach has the potential of being able to reuse the existing plethora of protocols for different fault-models and pull them into a single system design that gets the best-of-all-worlds performance.

6. Summary

This chapter has studied several design methodologies for the design of distributed systems, with focus on distributed storage structures. We first saw a taxonomy of design methodologies. Then we saw how declarative programming models can be used for the design of distributed p2p systems. Then, we have seen how natural phenomena and results and ideas from other fields of Science can be translated in a systematic manner into useful and practical distributed protocols. We also saw a few other design methodologies - both formal and informal.

The number and power of design methodologies for distributed systems is growing, and is reaching critical mass. Design methodologies have the potential to speed up the process of protocol design without stifling the creativity of this activity, and will likely play an increasing role in the next few years. Some methodologies are informal, others can be tied to automated code generation engines that allow a designer to go from idea to system implementation and experiments within the matter of a few seconds! If some of these methodologies (e.g., P2 being a case in point) had been invented a few years ago, the p2p research field would have had a much more accelerated development! Yet, like most research in this area, we end with an open-ended observation that the discovery of methodologies itself is a difficult task - there is, as yet, no "methodology to discover or reason about methodologies".

References

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

[2] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proc. 22nd ICDCS*, 2002.

[3] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Proc. ACM PODC*, 2003.

[4] B. Bershad and S. Savage et al. Extensibility, safety and performance in the SPIN operating system,. In *Proc. ACM SOSP*, pages 267–284, Dec. 1995.

[5] S. Camorlinga and K. Barker. The emergent thinker. In *Proc. SELF-STAR*, May-Jun. 2004.

[6] Computing Research Association (CRA). Grand Research Challenges in Distributed Systems. http://www.cra.org/reports/gc.systems.pdf.

[7] E. Gamma, R. Helm, R. Johnson, and Vlissides J. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1st edition, 1995.

[8] B. Garbinato and R. Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In *Proc. USENIX Conf. Obj.-Or. Tech. and Sys.*, pages 221–232, Jun. 1997.

[9] T.G. Griffin and J.L. Sobrinho. Metarouting. In *Proc. ACM SIGCOMM*, 2005.

[10] I. Gupta. *Building Scalable Solutions to Distributed Computing Problems using Probabilistic Components.* PhD Thesis, Dept. of Computer Science, Cornell University, Jan. 2004.

[11] I. Gupta. On the design of distributed protocols from differential equations. In *Proc. ACM PODC*, pages 216–225, 2004.

[12] I. Gupta and et al. A case for methodology research in self-* distributed systems. *LNCS 3460, Self-Star Properties in Complex Information Systems*, pages 260–272, 2005.

[13] A. Iamnitchi and I. Foster. Peer-to-peer approach to resource location in Grid environments. *Grid Resource Management*, 2003.

[14] E. Kohler and R. Morris et al. The Click modular router. *ACM TOCS*, 18(3), Aug. 2000.

[15] B.T. Loo, T. Condie, J. Hellerstein, and et al. Implementing declarative overlays. In *Proc. ACM SOSP*, 2005.

[16] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sep. 1989.

[17] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, Mary Baker, and Yanto Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. ACM SOSP*, pages 44–59, 2003.

[18] G. S. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *Proc. ACM PODC*, pages 197–205, 2004.

[19] R. V. Morales and I. Gupta. Providing both scale and security through a single core probabilistic protocol. In *Proc. 1st StoDiS*, 2005.

[20] A. Spector. Plenary Talk. ACM SOSP, 2003.

[21] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, pages 149–160, 2001.

[22] A. Tanenbaum. Plenary Talk. ACM SOSP, 2005.

[23] N. Thompson and I. Gupta. A composable methodology for proactive distributed protocols, Sept. 2004. TR UIUCDCS-R-2004-2490.

[24] R. van Renesse, S. Maffeis, and K. P. Birman. Horus: a flexible group communications system. *CACM*, 39(4):76–83, April 1996.

[25] J. Ventura, J. Rodrigues, and L. Rodrigues. Response time analysis of composable micro-protocols. In *Proc. 4th IEEE OORTDC*, pages 335–342, 2001.

[26] S.-H. Wu and S. A. Smolka et al. Composition and behaviors of probabilistic I/O automata. *TCS*, 176(1-2):1–38, Apr. 1997.

[27] J. Wylie and G.R. Goodson et al. A protocol family approach to survivable storage infrastructures. In *Proc. FUDICO*, 2004.

[28] G. Xie and J. Zhan et al. Routing design in operational networks: a look from the inside. In *Proc. ACM SIGCOMM*, pages 27–40, 2004.