The DHCP Failover Protocol: A Formal Perspective

Rui Fan[1], Ralph Droms[2], Nancy Griffeth[3], and Nancy Lynch[1]

[1] MIT CSAIL
[2] Cisco Systems
[3] Lehman College, CUNY

Abstract. We present a formal specification and analysis of a fault-tolerant DHCP algorithm, used to automatically configure certain host parameters in an IP network. Our algorithm uses ideas from an algorithm presented in [5], but is considerably simpler and at the same time more structured and rigorous. We specify the assumptions and behavior of our algorithm as traces of Timed Input/Output Automata, and prove its correctness using this formalism. Our algorithm is based on a composition of independent subalgorithms solving variants of the classical leader election and shared register problems in distributed computing. The modularity of our algorithm facilitates its understanding and analysis, and can also aid in optimizing the algorithm or proving lower bounds. Our work demonstrates that formal methods can be feasibly applied to complex real-world problems to improve and simplify their solutions.

1 Introduction

The *Dynamic Host Configuration Protocol (DHCP)* [4] is a widely deployed mechanism allowing devices to automatically obtain a unique IP address and other configuration information needed for communication on an IP network such as the Internet. Current implementations of DHCP use a single DHCP *server* to assign addresses from a predefined address pool. If the server fails, then addresses from the pool can no longer be reassigned, and are in effect lost from the address space. DHCP has recently been supplemented by the *DHCP Failover (DHCPF)* protocol [5], which manages an address pool using multiple servers. DHCPF increases the fault tolerance of DHCP, and also allows greater performance through load-balancing.

The main difficulty encountered in managing addresses using multiple servers instead of one is the need to maintain a consistent view across all the servers of the currently assigned addresses. Most standard database consistency techniques cannot be used to solve the DHCPF problem because they are too slow. A key insight of the algorithm described in [5] is to use two mechanisms for assigning addresses. The first mechanism relies on *synchronized clocks*; it is fast, requiring no communication, but limits how long addresses can be assigned. The second mechanism is slower, using explicit acknowledgments between the servers, but avoids the limitations on assignments. This algorithm is currently described in

J. Derrick and J. Vain (Eds.): FORTE 2007, LNCS 4574, pp. 211–226, 2007.

an Internet Draft that is over 130 pages long. Part of the length of the Draft is due to the need to deal with many possible types of concurrent server failures. The algorithm represents different combinations of failures as states of a system, and defines a large number of transitions between the states as failures occur or are resolved.

In this paper, we look at the DHCPF problem from a more formal and theoretical perspective. First, we extract the essential behavior of the algorithm in [5] and precisely specify the behavior as traces of a set of interacting *Timed I/O Automata (TIOA)* [6]. In this formulation, DHCPF is a kind of timed mutual exclusion problem. As in mutex, the safety condition requires that any address is used by at most one client at a time. The liveness condition requires that, under certain favorable timing conditions that are likely to occur in practice, any client wanting an IP address is granted one, as long as some addresses are available.

Our second contribution is to *decompose* this mutual exclusion problem into several simpler and independent subproblems, each mimicking a standard problem in distributed computing. In particular, we view DHCPF as involving the following two steps. First, for each IP address, we choose a *leader* server to be the only server allowed to assign that address. The leader for an address can change as different servers fail and recover, but we guarantee that there is at most one leader at a time. Thus, the first part of the problem can be seen as a multi-shot leader election problem. The second part of the problem consists of the leader assigning its address in such a way that even if it fails, and a different server becomes leader, the subsequent leader preserves the safety and liveness properties on that address. This can be seen as implementing a single-writer, multi-reader shared register, where the writer can change over time. In particular, only the current leader of an address is allowed to write assignments for the address to the register, but any server that takes over for a failed leader can read the register to ensure it does not double-allocate the address, and also gains the privilege to write to the register. The main idea that we adopt from [5] is to write to the register in two ways, either by an *implicit*, fast write, requiring no communication but relying on synchronized clocks, or by an *explicit*, slow write, using server acknowledgments.

There are several benefits to our formal treatment of DHCPF. First, the precise specification of DHCPF helps end-users of the service, who may need a rigorous understanding of the behavior of DHCPF that is difficult to obtain from the Internet Draft. Second, implementing DHCPF as a composition of smaller subalgorithms helps to understand and analyze its behavior, and also makes the algorithm easier to improve or optimize. For example, we can study the effects of tuning network parameters, such as the amount of clock skew or the bound on message delay, on the performance of our algorithm by studying their effects on the individual subalgorithms. We can also isolate the effects of different types of failures on the algorithm to how they affect the subalgorithms. This isolation is the main reason that our algorithm is simpler than the algorithm in [5]. Lastly, our decomposition suggests that it may be possible to prove lower bounds for

the DHCPF problem by proving lower bounds for the subproblems, which seems to be a considerably easier task.

Our treatment of DHCPF, while formal, was not mechanical. A considerable effort was involved in distilling the expansive description of DHCPF in the Internet Draft into a more concise formal specification. Nevertheless, parts of our specification are still more complicated than we would like. A second problem was finding a modular DHCPF algorithm, by matching parts of the specification against self-contained distributed computing problems. Systematizing this design process, indeed, formalizing the formalization process, would be a fascinating challenge.

The remainder of this paper is organized as follows. In Section 2, we describe the TIOA model. We give an overview of DHCPF and state the properties it satisfies in Section 3. We describe a DHCPF algorithm in Section 4, and prove its correctness and performance properties in Section 5. Finally, we conclude in Section 6.

2 Model and Notation

We model the clients, servers and communication network of our DHCPF algorithm as interacting Timed I/O Automata (*TIOA*). TIOA allows modeling of automata with continuous state spaces, whose executions evolve in real time. Our algorithm does not use the full power of this formalism. In what follows, we describe the TIOA model only to the extent necessary to understand our algorithm and its proof. Please see [6] for additional details. Each Timed I/O automaton has internal *state* variables, and discrete or continuous actions which change its state. We call discrete actions simply *actions*, and we call continuous actions *trajectories*. Actions always occur instantly, while a trajectory may have a positive time duration. As an example, we can model a mobile robot by a TIOA. The state represents the position of the robot. Trajectories are movements of the robot, and actions are changes in its destination (which we imagine as involving an instantaneous computations).

Several TIOAs can be *composed*. Roughly speaking, this forms a new automata whose state space is a Cartesian product of the state spaces of the constituent automata, and whose action space is the union of the constituent action spaces. However, certain states and actions become identified in the composition process; we describe this in more detail later. An *execution* is a sequence of the form $\alpha = \gamma_0\sigma_1\gamma_1\sigma_2\ldots\gamma_n\sigma_n$. Here, each γ_i represents a trajectory, and each σ_i represents a (discrete) action. We say a *state occurrence* (resp., *action occurrence*) is a *particular instance* of a state (resp., action) which occurs in an execution. Note that this is different from the state or action itself, which can occur multiple times in an execution. Let α be an execution, and let s, s' be state occurrences in α. We write $s \prec s'$ if s occurs before s' in α. We define $s \preceq s'$ in the obvious way; we also extend this notation to action occurrences σ, σ' in α. We write $s'' \in [s, s']$ if s'' is a state occurrence in α, and $s \preceq s'' \preceq s'$. Lastly, we say the *time* at which a state occurrence s occurs is the sum of the time durations of all the trajectories before s. We write this as $\zeta(s)$.

To model the clients in DHCPF, let C be an index set representing an arbitrary set of client processes. We use the notation i, i', i_1, etc. throughout the paper to denote clients. Similarly, let S be an index set representing server processes; we use j, j', j_1, etc. to denote servers. Each server can *fail* or *recover*. When a server fails, it stops performing any actions or trajectories. When it recovers, all its internal state variables are set to default values, and it begins executing from its initial state. We do not consider malicious server behaviors. Let Φ denote an arbitrary set of IP addresses; we write $\phi \in \Phi$ for a particular IP address. Servers will allocate addresses from Φ to clients. Each client and server is equipped with a real valued monotonically nondecreasing *clock* variable, which intuitively represents that process's perception of real time[1]. For the remainder of this paper, fix an arbitrary $\Delta \in \mathbb{R}^{\geq 0}$. We assume the *clock* of any process differs from real time by at most Δ. That is, we assume

Assumption 1. *Let $k \in C \cup S$. Then for any state occurrence s in any execution, we have $|s.clock_k - \zeta(s)| \leq \Delta$.*

Clients and servers communicate over a point-to-point message passing network. We assume that the network may lose, duplicate or reorder messages, but does not generate spurious messages. The network works as follows. Let $k, k' \in C \cup S$ be any two processes. When k wants to send a message m to k' during some action σ, we say that k *adds* (m, k') *to* $buffer$. If m is not lost by the network, then after a finite but nondeterministic time representing the message delay, the action $\mathsf{recv}_{k,k'}(m)$ occurs, causing k' to receive m; furthermore, k' knows that k sent the message. Note that these notational conventions are adopted from [6]. In describing our DHCPF algorithm in Section 4, we assume a network service exists which implements these communication actions.

3 A Formal Specification of DHCPF

In this section, we formally define the DHCPF problem. In particular, we define the interface between clients and servers in DHCPF. We also define the assumptions DHCPF makes about its operating environment. Finally, we define the properties DHCPF satisfies, given the environmental assumptions, in terms of the traces of a TIOA. In Section 4, we describe an algorithm satisfying this specification.

3.1 The DHCPF Interface

Figure 1 shows the client/server interface in DHCPF. It mimics, except for superficial differences, the client/server interface of the non-fault-tolerant DHCP. This is done to make the use of DHCPF instead of DHCP transparent to clients, in order to facilitate its deployment.

[1] Note that *clock* evolves according to a trajectory. In fact, *clock* is the only variable in our algorithm whose value follows a trajectory; the values of all other variables only change by (discrete) actions.

We will describe the interface of DHCPF by describing its typical modes of operation. DHCPF works by leasing IP addresses to clients. That is, a server tells a client that it can use a certain address up to some *lease* time, after which the client is supposed to release the address. There are two main types of interactions in DHCPF. When a client does not have a lease, it tries to *request* a lease. If the client already has a lease, it can try to *renew* the lease. Each type of interaction requires sending and receiving multiple messages. It is helpful to be able to identify all the messages in an interaction. We do this by labeling all the messages sent during the interaction by an *interaction instance* κ. Any two different interactions (even by the same client) are labeled with different κ. This labeling can be achieved using standard timestamping techniques [7,3]. We let K denote the set of all interaction instances.

We now describe the interaction for a client i to request a lease. Please also see Figure 1. Client i first broadcasts a discover message to all the servers, labeled by some interaction instance κ. A server j that receives the discover message sends an offer message to i for some address $\phi \in \Phi$[2]. Note that j must offer i an address immediately (if any are available). That is, the DHCP (and hence DHCPF) specification does not give j time to first communicate with the other servers to find out the current lease times for all addresses, before deciding what address to offer to i. It is precisely this need for an immediate response by j that prevents most database algorithms from being used to implement DHCPF. i may get offers for several ϕ's from different servers. i chooses one such ϕ as its preferred IP address, and broadcasts a request to lease that ϕ until time τ. Some server then responds to i with an ack message, leasing ϕ to i until time τ', where τ' may be *different* from the lease time τ which i requested. i is supposed to release ϕ when i's *clock* variable equals τ'.

If i already has a lease for ϕ until time τ', then i can try to renew its lease. To do this, i broadcasts a renew message to all the servers, including in the message the values of ϕ, τ' and τ, where τ is the new lease time that i wants. The message is labeled by κ. Some server then responds to i with an ack message extending i's lease on ϕ to time τ'', where τ'' may be different from i's desired lease τ.

In this paper, we assume that clients behave *correctly*. In particular, we assume that clients follow the order of interaction described above to request or renew an address. We also assume that clients release an IP address after their lease for the address expires. In general however, the servers have little means to enforce, or sometimes to even detect such behaviors. We leave the task of dealing with faulty clients as interesting future work.

Lastly, for any $j \in S$, we model server j's (stop) failure and recovery via the fail_j and $\mathsf{recover}_j$ actions.

3.2 DHCPF Assumptions

In this section, we describe the assumptions that DHCPF makes about its environment. The safety and liveness properties of DHCPF rely on different assumptions.

[2] If all the addresses in Φ are already offered or leased to other clients, then the server does not send offer.

$\mathsf{bcast}_i(\langle \mathsf{discover}, \kappa \rangle)$	i looks for an IP address; κ is the interaction instance.
$\mathsf{recv}_{i,j}(\langle \mathsf{discover}, \kappa \rangle)$	j receives i's discover message.
$\mathsf{send}_{j,i}(\langle \mathsf{offer}, \kappa, \phi \rangle)$	j offers ϕ to i.
$\mathsf{recv}_{j,i}(\langle \mathsf{offer}, \kappa, \phi \rangle)$	i receives j's offer.
$\mathsf{bcast}_i(\langle \mathsf{request}, \kappa, \phi, \tau \rangle)$	i requests ϕ till time τ.
$\mathsf{recv}_{i,j}(\langle \mathsf{request}, \kappa, \phi, \tau \rangle)$	j receives i's lease request.
$\mathsf{bcast}_i(\langle \mathsf{renew}, \kappa, \phi, \tau, \tau' \rangle)$	i wants to renew ϕ till time τ; τ' is i's last lease time for ϕ.
$\mathsf{recv}_{i,j}(\langle \mathsf{renew}, \kappa, \phi, \tau, \tau' \rangle)$	j receives i's renew message.
$\mathsf{send}_{j,i}(\langle \mathsf{ack}, \kappa, \phi, \tau \rangle)$	j gives i address ϕ till time τ.
$\mathsf{recv}_{j,i}(\langle \mathsf{ack}, \kappa, \phi, \tau \rangle)$	i receives j's acknowledgment.
$\mathsf{fail}_j, \mathsf{recover}_j$	j fails or recovers.

Fig. 1. The DHCPF protocol interface, for $i \in C, j \in S$

The DHCPF safety property roughly says that any IP address is leased to at most one client at a time. To satisfy this property, DHCPF requires a *failure detector*, which is a service telling every server which other servers have failed. The DHCPF liveness properties says that when a client requests or tries to renew an address, it will get an address within a few message round trips' time, as long as some addresses are available. This property is only satisfied in "nice" periods of an execution. These assumptions are described in the proceeding sections.

An effort was made to "minimize" the assumptions that DHCPF relies upon. Indeed, at an intuitive level, it seems unlikely that any DHCPF algorithm can work correctly if servers have no idea about each others' status, if servers continuously fail and recover, or if the network delays messages for very long times. Furthermore, we believe that the assumptions we make are sufficiently weak that they are likely to be satisfied, at least typically, in practice. Finally, while we do not study questions related to minimality or impossibility in this paper, we believe that these may be interesting future work.

In the remainder of this section, let α be an arbitrary execution. All state and action occurrences are assumed to occur in α.

A Failure Detector Service. Recall that each server can fail, and then subsequently recover. We define the following.

Definition 3.1. *Let $j \in S$ and let s be a state occurrence. We say j is* alive *in s if there exists an action occurrence $\sigma = \mathsf{recover}_j$ such that $\sigma \prec s$, and for all action occurrences σ' such that $\sigma \prec \sigma' \prec s$, we have $\sigma' \neq \mathsf{fail}_j$. If j is not alive in s, we say j is* dead *in s.*

We assume that all servers are initially alive.

Definition 3.2. *Let s and s' be state occurrences, with $s \prec s'$. We let $\Lambda(s, s') = \{j \mid (j \in S) \wedge (\forall s'' \in [s, s'] : j \text{ is alive in } s'')\}$ be the set of servers that are alive throughout the interval $[s, s']$.*

A *failure detector* service Υ informs each server which other servers are alive or dead. In practice, Υ might represent a system administrator who manually

informs servers about failures and recoveries. Formally, we assume that for each $j \in S$, in addition to j's actions shown in Figure 1, j also has the following two sets of actions, which we call *FD-actions*.

1. $\forall j' \in S : \mathsf{recv}_{\Upsilon,j}(\langle \mathsf{dead}, j' \rangle)$. Υ informs j that j' is dead.
2. $\forall j' \in S : \mathsf{recv}_{\Upsilon,j}(\langle \mathsf{alive}, j' \rangle)$. Υ informs j that j' is alive.

In order to be useful, Υ is required to be *accurate* and *timely*. In particular, let λ be some nonnegative constant. The accuracy property says that if a server j' has been alive or dead for λ or more time before the current time, then any information Υ gives to a server j about j' is correct. The timeliness property says that if j is alive for at least λ time, then j will receive failure information from Υ about every server $j' \in S$. These are captured in the following definition.

Definition 3.3. *Let $\lambda \in \mathbb{R}^{\geq 0}$. We say a failure detector Υ is* λ-perfect *if the following hold for any state occurrences s and s' such that $\zeta(s') - \zeta(s) \geq \lambda$.*

1. (Accuracy) *Let $j, j' \in S$, and suppose the action occurrence $\mathsf{recv}_{\Upsilon,j}(\langle \mathsf{dead}, j' \rangle)$ (resp., $\mathsf{recv}_{\Upsilon,j}(\langle \mathsf{alive}, j' \rangle)$) immediately precedes s'. Then j' is dead (resp., alive) in some state during $[s, s']$.*
2. (Timeliness) *Suppose $j \in \Lambda(s, s')$. Then for every $j' \in S$, either $\mathsf{recv}_{\Upsilon,j}(\langle \mathsf{fail}, j' \rangle) \in [s, s']$ or $\mathsf{recv}_{\Upsilon,j}(\langle \mathsf{recover}, j' \rangle) \in [s, s']$.*

In order to guarantee correct behavior, the DHCPF algorithm we describe in Section 4 requires that a λ-perfect Υ, for some finite λ. In [5], a weaker failure detector is used which can sometimes give incorrect information. However, in such cases, the algorithm of [5] can actually allocate the same IP address to more than one client. We believe that this limitation is inherent. That is, we believe (though we do not prove) that any fault-tolerant algorithm implementing a reasonable form of DHCP requires the use of a server failure detector satisfying similar safety and liveness properties to those we define above. Indeed, failure detectors are a widely adopted notion in distributed computing, and are provably necessary to solve many problems, especially *agreement problems* of the type similar to DHCPF; see e.g. [1,2].

Stable and Timely Periods. We now describe the assumptions DHCPF makes in order to satisfy its liveness properties. As mentioned earlier, these properties only hold during "nice" periods of an execution. A nice period roughly corresponds to a sufficiently long time interval in which no servers fail or recover, and in which messages are delivered quickly. More precisely, we define the following.

Definition 3.4. *Let $\lambda \in \mathbb{R}^{\geq 0}$, and let s and s' be state occurrences with $s \prec s'$. We say that $[s, s']$ is* λ-stable *if we have the following*

1. *There exists $j \in S$ such that j is alive in state occurrence s'', $\forall s'' : \zeta(s) - \lambda \leq \zeta(s'') \leq \zeta(s')$.*
2. *For all action occurrences σ such that $\zeta(s) - \lambda \leq \zeta(\sigma) \leq \zeta(s')$, we have $\sigma \notin \{\mathsf{fail}_*, \mathsf{recover}_*\}$.*

Thus, $[s, s']$ is λ-stable if in the entire time duration $[\zeta(s) - \lambda, \zeta(s')]$, no servers fail or recover, and there is at least one live server.

Definition 3.5. *Let s and s' be state occurrences such that $s \prec s'$, and let $\lambda \in \mathbb{R}^{\geq 0}$ be such that $\lambda \leq \zeta(s') - \zeta(s)$. We say $[s, s']$ is* λ-timely *if for any message m, for any $k, k' \in C \cup S$, and for any action occurrence σ adding (m, k') to buffer, such that $\zeta(s) \leq \zeta(\sigma) \leq \zeta(s') - \lambda$, there exists action occurrence $\sigma' = \mathsf{recv}_{k,k'}(m)$, such that $\sigma \prec \sigma' \preceq s'$.*

Thus, $[s, s']$ is λ timely if the interval is at least λ in duration, and any message sent during the interval at least λ time before s' is received during $[s, s']$.

3.3 DHCPF Properties

In this section, we state the properties that DHCPF guarantees, under the assumptions of Section 3.2. We first define the following.

Definition 3.6. *Let $i \in C$, $\phi \in \Phi$, and let s be a state occurrence.*

1. *We say i* owns *ϕ in s if there exists an action occurrence $\sigma = \mathsf{send}_{*,i}(\langle \mathsf{ack}, *, \phi, \tau \rangle)$ such that $\sigma \prec s$, and $\zeta(s) \leq \tau + \Delta$.*
2. *We let $\omega(s, \phi) = \{i \mid (i \in C) \wedge (i \text{ owns } \phi \text{ in } s)\}$.*

Thus, i owns ϕ in s if i has been sent an acknowledgment before state s to lease ϕ until time τ, and s happens at or before time $\tau + \Delta$. Intuitively, the Δ in the definition is to account for the fact that, when i is given a lease on ϕ for time τ, i may not release ϕ until real time $\tau + \Delta$, due to i's clock skew.

The following definition describes the properties satisfied by the DHCPF protocol. The safety property states that at most one client owns any IP address at a time. The request and renew liveness properties are complicated to state. But intuitively, they simply say that in nice time periods in which servers do not fail or recover, messages are delivered quickly, and not all IP addresses have already been allocated, a client always succeeds in quickly requesting or renewing an address. The liveness properties are described in more detail following Definition 3.7.

Definition 3.7. *Let $\nu, \delta \in \mathbb{R}^{\geq 0}$. Suppose Υ is a ν-perfect failure detector. Then an algorithm $\mathcal{A}$ satisfies the* DHCPF *protocol if $\mathcal{A}$'s external actions includes the actions shown in Figure 1, and for every execution α of $\mathcal{A}$, the following properties hold.*

1. **Safety:** *For any $\phi \in \Phi$ and any state occurrence s, we have $|\omega(s, \phi)| \leq 1$.*
2. **Request Liveness:** *Let $i \in C$, $\kappa \in K$, and let s and s' be state occurrences, with $s \prec s'$. Let $\sigma = \mathsf{bcast}_i(\langle \mathsf{discover}, \kappa, * \rangle)$. Let $\sigma_j = \mathsf{recv}_{j,i}(\langle \mathsf{discover}, \kappa, *, \rangle)$, $\forall j \in S$. Let $\sigma_\phi = \mathsf{send}_{*,i}(\langle \mathsf{offer}, \kappa, \phi, * \rangle), \forall \phi \in \Phi$. Suppose that $[s, s']$ is $(4\nu + 4\Delta)$-stable and δ-timely, and $\zeta(s') - \zeta(s) \geq 4\delta$. Also suppose that $\sigma \in [s, s']$, and $\zeta(\sigma) \leq \zeta(s') - 4\delta$. Then there exists $\xi_1, \xi_2 \in \mathbb{R}^{\geq 0}$ such that the following hold.*

(a) *For every* $j \in \Lambda(s, s')$, *we have* $\sigma_j \in [s, s']$, *and* $\zeta(\sigma_j) \leq \zeta(\sigma) + \delta$. *Let* s_j *be the state occurrence immediately following* σ_j, $\forall j \in \Lambda(s, s')$.

(b) *Either there exists* $\phi \in \Phi$ *such that* $\sigma_\phi \in [s, s']$ *and* $\zeta(\sigma_\phi) \leq \zeta(\sigma) + 2\delta$, *or for every* $\phi \in \Phi$, *there exists* $j_\phi \in S$ *such that one of the following holds.*

i. Let $\sigma^1_\phi = \mathsf{send}_{j_\phi,*}(\langle \mathsf{offer}, *, \phi, *, \rangle)$ *and* $\sigma^2_\phi = \mathsf{recv}_{*,j_\phi}(\langle \mathsf{request}, *, \phi, * \rangle)$. *We have* $\sigma^1_\phi \prec \sigma_{j_\phi}$, $\zeta(\sigma^1_\phi) \geq \zeta(\sigma_{j_\phi}) - \xi_1 - 2\Delta$, *and* $\sigma^2_\phi \not\prec \sigma_{j_\phi}$.

ii. Let $\sigma^3_\phi = \mathsf{recv}_{*,*}(\langle \mathsf{request}, *, \phi, \tau_\phi \rangle)$. *We have* $\sigma^3_\phi \prec \sigma_{j_\phi}$, *and* $\zeta(\sigma_{j_\phi}) \leq \max(\tau_\phi + 3\Delta, \zeta(\sigma^3_\phi) + \xi_2 + 3\Delta)$.

iii. Let $\sigma^4_\phi = \mathsf{recv}_{*,*}(\langle \mathsf{renew}, *, \phi, \tau'_\phi, * \rangle)$. *We have* $\sigma^4_\phi \prec \sigma_{j_\phi}$, *and* $\zeta(\sigma_{j_\phi}) \leq \max(\tau'_\phi + 3\Delta, \zeta(\sigma^4_\phi) + \xi_2 + 3\Delta)$.

(c) *Let* $\Phi' \subseteq \Phi$, *and suppose* $\forall \phi \in \Phi' : \sigma_\phi \in [s, s']$. *Let* $\sigma'_\phi = \mathsf{recv}_{*,i}(\mathsf{ack}, \kappa, \phi, *)\rangle), \forall \phi \in \Phi$. *Then there exists* $\phi \in \Phi'$ *such that* $\sigma'_\phi \in [s, s']$ *and* $\zeta(\sigma'_\phi) \leq \zeta(\sigma) + 4\delta$.

3. **Renew Liveness:** *Let* $i \in C$, $\kappa \in K$, $\tau, \tau' \in \mathbb{R}^+$, $\phi \in \Phi$, *and let* s *and* s' *be state occurrences with* $s \prec s'$. *Let* $\sigma = \mathsf{send}_{*,i}(\langle \mathsf{ack}, *, \phi, \tau' \rangle)$, $\sigma' = \mathsf{bcast}_i(\langle \mathsf{renew}, \kappa, \phi, \tau, \tau' \rangle)$, *and* $\sigma'' = \mathsf{recv}_{*,i}(\langle \mathsf{ack}, \kappa, \phi, * \rangle)$. *Suppose* $[s, s']$ *is* $(4\nu + 4\Delta)$*-stable and* δ*-timely, and* $\zeta(s') - \zeta(s) \geq 2\delta$. *Also, suppose* $\sigma \in [s, s']$, *with* $\sigma \prec \sigma'$, $\zeta(\sigma') \leq \zeta(s') - 2\delta$ *and* $\zeta(\sigma') \leq \tau' - \delta - \Delta$. *Then* $\sigma'' \in [s, s']$ *and* $\zeta(\sigma'') \leq \zeta(\sigma') + 2\delta$.

We now describe conditions 2 and 3 in more detail. Intuitively, the request liveness property says that a client that requests an IP address will get one quickly, unless there is some "excuse" not to give it one. Specifically, let $[s, s']$ be an interval that is at least 4δ time long, and is stable and timely. Then if a client i broadcasts a discover message at time $t = \zeta(\sigma)$, its message is received by all live servers no later than time $t + \delta$. Condition 2.*b* states that either i receives an offer for some IP address ϕ, or the servers have an excuse not to offer i any address; conditions 2.*b.i* – 2.*b.iii* list various excuses not to offer i address ϕ. In condition 2.*b.i*, ϕ has recently been offered to some client, but has not been requested. Thus, ϕ is reserved for the other client. In 2.*b.ii*, some client requested ϕ for time τ_ϕ in action occurrence σ^3_ϕ. The quantity $\max(\tau_\phi + 3\Delta, \zeta(\sigma^3_\phi) + \xi_2 + 3\Delta)$ represents a lease for ϕ that was potentially given out to that client[3]. If $\zeta(\sigma_{j_\phi}) \leq \max(\tau_\phi + 3\Delta, \zeta(\sigma^3_\phi) + \xi_2 + 3\Delta)$, then i's discover message arrived at server j_ϕ before the last (potential) lease for ϕ has expired, which justifies i not being offered ϕ. Condition 2.*b.iii* is similar to 2.*b.ii*, but deals with a renew on ϕ by another client. Lastly, condition 2.*c* says that if i is offered some IP addresses, then i will also be given a lease for some such address no later than time $t + 4\delta$. The renew liveness condition says that in a stable and timely interval, if client i tries to renew ϕ sufficiently long before its previous lease τ' on ϕ expires, then i will be granted a new lease on ϕ. Finally, note that despite their complicated statement, the excuses in the liveness property are in some ways inherent to the DHCPF problem. Nevertheless, it would be desirable to find a more succinct way of expressing them.

[3] The ξ_2 and Δ terms represent some "slack" in the estimate for the potential lease.

4 A DHCPF Algorithm

In this section, we describe an algorithm satisfying the DHCPF specification in Definition 3.7. Our algorithm uses ideas described in [5], and also introduces several new ones. Compared to [5], our algorithm is more structured, and is considerably simpler to understand and analyze. The algorithm is based on a decomposition of the DHCPF protocol into two subproblems, with the goal to base the subproblems on well-studied problems in distributed computing, and to maximize the amount of "independence" between the subproblems. In the first problem, we find, for each address $\phi \in \Phi$, a server which we call the *leader* for ϕ. The leader for ϕ is the only server that is allowed to lease ϕ to the clients. The leader for ϕ may change during an execution, as servers fail and recover. However, we will ensure that at all times, there is at most one leader for ϕ. We call this the *leader election* problem. Given a leader for ϕ, say $j \in S$, the second problem involves j leasing ϕ to the clients in a way such that even if j fails, and another server j' takes over as leader for ϕ, the leases given out by j' for ϕ will not conflict with leases given out by j. We call this the *lease* problem. In the remainder of this section, we first describe an algorithm to solve the leader election problem, then give an algorithm which uses the leader election algorithm to solve the lease problem. Our DHCPF algorithm, satisfying the properties in Definition 3.7, is the (formal) composition of these two algorithms.

4.1 Leader Election Algorithm

We now present the *Elect* algorithm for solving the leader election problem. We first describe the algorithm, then prove the properties it satisfies in Theorems 4.2 and 4.3. For the remainder of this section, fix an arbitrary $\nu \in \mathbb{R}^{\geq 0}$. *Elect* uses a ν-perfect failure detector Υ. Recall that Δ is a bound on the maximum clock skew of any server (or client).

The pseudocode for server j running the $Elect_j$ algorithm is shown in Figure 2. For each $\phi \in \Phi$, let $<_\phi$ be a total ordering on the set S. If $S' \subseteq S$, then $\min_\phi S'$ denotes the minimum server in S', with respect to ordering $<_\phi$. The

```
input recv_{Υ,j}(⟨dead, j'⟩)
Effect:
    live ← live \ {j'}
    for every φ ∈ Φ do
     if ((j = min_φ live) ∧
         (clock ≥ rec-time + 2ν + 2Δ)) then
      leader ← leader ∪ {φ}
      lead-time[φ] ← clock
      lead[φ] ← true

input alive_j
Effect:
    rec-time ← clock

input recv_{Υ,j}(⟨alive, j'⟩)
Effect:
    live ← live ∪ {j'}
    for every φ ∈ Φ do
     if (j ≠ min_φ live) then
      leader ← leader \ {φ}

output lead_j(φ)
Precondition:
    lead[φ] = true
Effect:
    lead[φ] ← false
```

Fig. 2. The $Elect_j$ algorithm, for $j \in S$

idea of *Elect* is to let the $\min_\phi$ *live server* be the leader for ϕ[4]. Information about which servers are alive is provided to j by Υ; j keeps track of the servers it thinks are alive in the set *live*, which initially equals S. j keeps track of the IP addresses for which it is the leader in the set *leader*[5]; *leader* initially equals $\emptyset$. *lead*$[\phi]$ is a helper variable to flag when j becomes the leader for ϕ. When j recovers from a failure, it stores the time of its recovery in *rec-time*. Whenever j receives an alive message about server j' from Υ, j adds j' to *live*. After this, if for any $\phi \in \Phi$, j is no longer the $\min_\phi$ live server, it removes ϕ from *leader*. When j receives a dead message for j' from Υ, j removes j' from $fail$. Then, if j becomes the $\min_\phi$ server for ϕ, and if j's current time is sufficiently larger than j's last recovery time, j becomes leader for ϕ, by adding ϕ to *leader*. j also records the time it becomes leader in *lead-time*$[\phi]$.

Correctness of *Elect*. Before stating the correctness properties *Elect* satisfies, we first define the following.

Definition 4.1. *Let $\phi \in \Phi$, and let s be any state occurrence. We say $\Omega(s, \phi) = \{j \mid (j \in S) \wedge (\phi \in s.leader_j)\}$ is the set of* leaders *for ϕ in s.*

Recall that for state occurrences $s \prec s'$, $\Lambda(s, s')$ is the set of servers that are alive throughout the interval $[s, s']$, and $\zeta(s)$ is the real time at which s occurs.. The following safety property states that for any address ϕ, there is at most one server that is the leader for ϕ at any time. Due to lack of space, we omit the full proof of the theorem; it appears in the full version of this paper.

Theorem 4.2 (Safety). *For any execution α of* Elect*, any state occurrence s in α, and any $\phi \in \Phi$, we have $|\Omega(s, \phi)| \leq 1$.*

Proof. The basic idea is that each server waits for a period of time after it recovers before trying to become leader. During that time, because Υ is timely, the server will be able to hear about any other live servers which might be competing to become leader. Thus, all candidates to become leader will know about each other, and so only the minimum one will be elected leader. □

The following liveness property says that in any sufficiently stable state occurrence, for any $\phi \in \Phi$, the $\min_\phi$ live server is the leader for ϕ. The proof appears in the full paper. The basic idea is that in a stable execution, all the live servers know about each other, and so the minimum live server is elected leader.

Theorem 4.3 (Liveness). *Let α be any execution, and let s be any state occurrence such that s is $(4\nu + 4\Delta)$-stable. Then for all $\phi \in \Phi$, we have $\phi \in s.leader_{\min_\phi \Lambda(s,s)}$.*

[4] Note that the reason we use a (possibly) different ordering $<_\phi$ for each ϕ is for load-balancing. Indeed, we can define a canonical ordering $<$ on S and let the minimum (w.r.t. $<$) live server be the leader for every IP address; but this may overload the minimum live server while the other servers do nothing.

[5] Note that j can be the leader for several addresses at the same time.

4.2 Lease Algorithm

In this section, we describe the lease algorithm, uncreatively named *Lease*. *Lease* uses *Elect*; in particular, every server $j \in S$ running *Lease* needs to know $lead\text{-}time[\phi]_j$, for all $\phi \in \Phi$, and also needs to know $leader_j$. That is, j needs to know when it last became the leader for ϕ (if ever), and what addresses it is leader for. The *Lease* algorithm is shown in Figure 3. We first describe the algorithm, then prove the properties it satisfies in the next section.

Consider any $\phi \in \Phi$, and suppose j is the current leader for ϕ. The main thing *Lease* needs to ensure is that when j gives out a lease for ϕ, the other servers know about this lease in some way, so that if j later fails, the next leader for ϕ will not give out a conflicting lease. To let other servers know about its leases, j gives out *two types* of leases: an (intuitively, short)*Minimum Client Lead Time (MCLT) lease*, and an (intuitively, long) *acknowledged lease*. For the remainder of this paper, we fix a constant $\mu \in \mathbb{R}^+$ which we call the *MCLT value*. Roughly speaking, when a client i sends j a request message to lease ϕ until time τ, j first gives i a lease equal to j's current clock value plus μ. This is the MCLT lease; note that it may be less than τ. Immediately after acknowledging the client, j broadcasts a potlease-write message to all the servers containing ϕ and τ. When a server j' receives this message, it sets $potlease[\phi]_{j'} \leftarrow \tau$; now, j' knows that some client has requested a lease of τ on ϕ. j' also acknowledges j with a potlease-write-ack message. When j receives acks about ϕ and τ from *every server* in S, j sets $acklease[\phi]_j \leftarrow \tau$. Now, if i sends a renew message for ϕ for time τ', j will give i a lease for time τ; this is an acknowledged lease. Thus intuitively, j begins by giving i a "temporary" MCLT lease, intended to tide i over while j negotiates a "real" acknowledged lease for i with the other servers.

We now describe the *Lease* algorithm for a server j in more detail, keeping in mind the above schema. In j's initial state, we set $reserved_j = \emptyset$, $potlease[\phi]_j =$

```
input lead_j(φ)
Effect:
    potlease[φ] ← max(lead-time[φ]+
        μ + 2Δ, potlease[φ])

input recv_{i,j}(⟨discover, κ, τ⟩)
Effect:
    S ← {φ | (φ ∈ leader) ∧ ((∗, φ, ∗) ∉ reserved)∧
        (potlease[φ] + 2Δ < clock)}
    if S ≠ ∅ then
      choose φ ∈ S
      reserved ← reserved ∪ {(κ, φ, clock)}
      add (⟨offer, κ, φ, 0⟩, i) to buffer

input recv_{i,j}(⟨request, κ, φ, τ⟩)
Effect:
    reserved ← reserved\(κ, ∗, ∗)
    if (φ ∈ leader) ∧ (potlease[φ] + 2Δ < clock) then
      acklease[φ] ← clock + μ
      τ ← max(τ, acklease[φ])
      potlease[φ] ← acklease[φ]
      add (⟨ack, κ, φ, acklease[φ]⟩, i) to buffer
      for every j′ ∈ S do
        add (⟨potlease-write, φ, κ, τ⟩, j′) to buffer

input recv_{i,j}(⟨renew, κ, φ, τ, τ′⟩)
Effect:
    if (φ ∈ leader) ∧ (τ′ ≥ clock) then
      acklease[φ] ← max(clock + μ, acklease[φ])
      τ ← max(τ, acklease[φ])
      potlease[φ] ← max(acklease[φ], potlease[φ])
      add (⟨ack, κ, φ, acklease[φ]⟩, i) to buffer
      for every j′ ∈ S do
        add (⟨potlease-write, φ, κ, τ⟩, j′) to buffer

input recv_{j′,j}(⟨potlease-write, φ, κ, τ⟩)
Effect:
    potlease[φ] ← max(τ, potlease[φ])
    add (⟨potlease-write-ack, φ, κ, τ⟩, j′) to buffer

input recv_{j′,j}(⟨potlease-write-ack, φ, κ, τ⟩)
Effect:
    write-acks[κ] ← write-acks[κ] ∪ {j′}
    if write-acks[κ] = S then
      acklease[φ] ← max(τ, acklease[φ])

input cleanup_j()
Effect:
    S ← {(κ, φ, t) | ((κ, φ, t) ∈ reserved)∧
            (t < clock − θ)}
    reserved ← reserved \ S
```

Fig. 3. The $Lease_j$ algorithm

$acklease[\phi]_j = 0, \forall \phi \in \Phi$, and $write\text{-}acks[\kappa]_j = \emptyset, \forall \kappa \in K$. To request a lease, a client i broadcasts a discover message. When j receives this message, it checks three things. First, j checks that it is the leader for ϕ, i.e. $\phi \in leader_j$. Then j checks that $\phi \notin reserved_j$; that is, no other client asked j for ϕ before i. Lastly, j checks that $potlease[\phi]_j + 2\Delta < clock_j$. $potlease[\phi]_j$ represents the j's estimate of the highest lease which could possibly have been given out for ϕ, by any server (e.g., by previous leaders for ϕ). If $potlease[\phi]_j + 2\Delta < clock_j$, then j knows that any previous leases for ϕ have definitely expired. If all three conditions hold, then j sends an offer for ϕ to i. j also adds i's interaction instance κ, along with ϕ and the current time, to $reserved_j$. Having received offers from possibly multiple servers, i sends a request message for its preferred address. If j receives a request message from i for ϕ with lease time τ, it again checks the above conditions. If they hold, then j sends i an MCLT lease, i.e., a lease equal to $clock_j + \mu$. j also sends potlease-write for ϕ and τ to all the servers. If j' receives j's potlease-write, it sets $potlease[\phi]_{j'} \leftarrow \max(\tau, potlease[\phi]_{j'})$, and acknowledges j with potlease-write-ack. j keeps track of which servers have acknowledged it in $write\text{-}acks[\kappa]_j$. When $write\text{-}acks[\kappa]_j = S$, j sets $acklease[\phi]_j \leftarrow \max(\tau, acklease[\phi]_j)$.

When a client i which currently has a lease time τ' for ϕ asks j to renew ϕ for time τ, j first checks that it is the owner for ϕ, and then that $\tau' \geq clock_j$. The latter condition checks that the current lease τ' has not yet expired, giving i the right to renew ϕ. If both conditions hold, j sets $acklease[\phi]_j \leftarrow \max(clock_j + \mu, acklease[\phi]_j)$, and sends i a lease for ϕ until time $acklease[\phi]_j$; thus, j gives i an acknowledged lease. j also broadcasts potlease-write for ϕ and τ to the other servers.

For the remainder of this paper, fix a constant $\theta \in \mathbb{R}^+$. The $\mathsf{cleanup}_j$ action removes addresses from $reserved_j$ that were offered at least θ time ago to some clients, but have not been requested. This is to reclaim addresses offered to clients that fail (or are slow) after being offered an address.

4.3 The Composed DHCPF Algorithm

We define our DHCPF algorithm to be the formal composition $\prod_j Elect_j \times Lease_j$. We refer to [6] for a full description of the composition operator $\times$. Briefly, $\times$ works by sharing the variables that the composed automata have in common, and identifying the output actions of one automaton with the input actions of the same name of another automaton. In our case, this means that for all $j \in S$, $Elect_j$ and $Lease_j$ both have access to the variables $leader_j$, and $lead\text{-}time[\phi]_j$, $\forall \phi \in \Phi$. $Elect_j$ and $Lease_j$ have only one type of action in common, $\mathsf{lead}_j(\phi)$, $\forall \phi \in \Phi$. By composing the algorithms, the input action $\mathsf{lead}_j(\phi)$ of $Lease_j$ is triggered whenever the output action $\mathsf{lead}_j(\phi)$ of $Elect_j$ occurs. Thus, $Elect_j$ notifies $Lease_j$ whenever j becomes the leader for ϕ. We will call $\prod_j Elect_j \times Lease_j$ the algorithm $\mathcal{C}$, for "composed".

5 Properties of $\mathcal{C}$

In this section, we show that the execution traces of $\mathcal{C}$ satisfy the DHCPF specification in Definition 3.7. In the remainder of this section, fix $\alpha = \gamma_0 \sigma_1 \gamma_1 \ldots \sigma_n \gamma_n$

to be an arbitrary execution of $\mathcal{C}$. Define s_k to be the state of $\mathcal{C}$ immediately before γ_k, for $k \in 0..n$. We first consider the safety properties.

5.1 Safety Properties of $\mathcal{C}$

The basic idea for showing that $\mathcal{C}$ never allocates the same IP address to more than one client is to show that the $potlease[\phi]$ value of the leader for ϕ is always an overestimate of the actual lease given out for ϕ. For example, suppose the last lease for ϕ was given out by server $j' \neq j$, at real time t. Then, if j' gave out an MCLT lease, the value of the lease is approximately $t + \mu$. If j becomes the leader for ϕ, at a time $t' > t$, its first step is to set its $potlease[\phi]$ value to at least $t' + \mu + 2\Delta$, which overestimates the real lease time. Otherwise, if the lease given out by j' was an acknowledged lease, then j received a potlease-write message for the lease from j', and thus also set its $potlease[\phi]$ value to be at least the value of j''s lease. Now, because the leader's $potlease[\phi]$ value is at least as large as the highest lease given out for ϕ, and because the leader checks that the current time on its clock (plus some slack) is larger than $potlease[\phi]$ before giving out a new lease for ϕ, then ϕ will never be double allocated. We now state a series of lemmas to formalize this idea. Due to lack of space, the complete proofs appear in the full paper. The proof method in most cases is an induction on the execution length. That is, we show a lemma holds in the initial state of the execution, and check that every step of the execution preserves the lemma. The lemmas were chosen so that these checks are typically quite straightforward. In fact, most of the proofs seem to be checkable by interactive theorem prover tools.

The first lemma states that $potlease[\phi]_j$ never decreases during α, for any j and ϕ.

Lemma 5.1. *Let $j \in S, \phi \in \Phi$, and let s and s' be state occurrences such that $s \prec s'$. Then $s.potlease[\phi]_j \leq s'.potlease[\phi]_j$.*

We define the following.

Definition 5.2. *Let $i \in C$, $\kappa \in K$, $\phi \in \Phi$, $\tau, \tau' \in \mathbb{R}^{\geq 0}$, and suppose there exists an action occurrence $\sigma = \mathsf{bcast}_i(\langle \mathsf{request}, \kappa, \phi, \tau \rangle)$ or $\sigma' = \mathsf{bcast}_i(\langle \mathsf{renew}, \kappa, \phi, \tau, \tau' \rangle)$ in α. Then we define $i_\kappa = i$, $\phi_\kappa = \phi$, and $\tau_\kappa = \tau$.*

Thus, i_κ, ϕ_κ, and τ_κ are the client, IP address and desired lease time associated with a request or renew interaction instance κ. Note that these are well defined because every interaction in α uses a different κ, so that at most one of σ or σ' can occur in α. The following lemma says that given an interaction instance κ and its associated ϕ_κ and τ_κ, if one server is contained in the $write\text{-}acks[\kappa]$ variable of another server, then the former server's $potlease[\phi_\kappa]$ is at least τ_κ.

Lemma 5.3. *Let $\kappa \in K$, $j, j' \in S$, and let s be a state occurrence. Suppose $j \in s.write\text{-}acks[\kappa]_{j'}$. Then $s.potlease[\phi_\kappa]_j \geq \tau_\kappa$.*

The next lemma compares the values of $potlease[\phi]$, $acklease[\phi]$ and $clock$ at different servers, for any ϕ. The basic proof idea is that j can estimate $acklease[\phi]_{j'}$

either through a potlease-write message from j' when j' gives out an acknowledged lease, or by adding μ to j's own clock, when j' gives out an MCLT lease. The 2Δ term accounts for the possible skew between j and j''s clocks.

Lemma 5.4. *Let $\phi \in \Phi$, $j, j' \in S$, and let s be a state occurrence. Then we have* $\max(s.potlease[\phi]_j, s.clock_j + \mu + 2\Delta) \geq s.acklease[\phi]_{j'}$.

The next lemma states that the *potlease*$[\phi]$ value of the leader for ϕ is at least as large as any *acklease*$[\phi]$ value. The proof uses Lemma 5.4, the fact that there is at most one leader for ϕ at a time (by Theorem 4.2), and the fact that j sets $potlease[\phi]_j$ to $\max(lead\text{-}time[\phi]_j + \mu + 2\Delta, potlease[\phi]_j)$ upon becoming leader for ϕ.

Lemma 5.5. *Let $\phi \in \Phi$, $j \in S$, and let s be a state occurrence. Then if $\Omega(s, \phi) \neq \emptyset$, we have $s.potlease[\phi]_{\Omega(s,\phi)} \geq s.acklease[\phi]_j$.*

The next lemma states that *acklease*$[\phi]$ never decreases, for any j and ϕ.

Lemma 5.6. *Let $j \in S, \phi \in \Phi$, and let s and s' be state occurrences such that $s \prec s'$. Then $s.acklease[\phi]_j \leq s'.acklease[\phi]_j$.*

Combining the above lemmas, the following theorem states that at most one client is assigned any IP address at any time. Intuitively, the theorem holds because the leader for ϕ has a good estimate of the maximum possible lease given out for ϕ using *potlease*$[\phi]$, and checks that this lease has expired before giving out a new lease for ϕ.

Theorem 5.7. *Any execution of $\mathcal{C}$ satisfies the safety property in Definition 3.7.*

5.2 Liveness Properties of $\mathcal{C}$

The next two theorems state that $\mathcal{C}$ satisfies the request and renew liveness conditions of the DHCPF specification. Recall that θ is the amount of time a server reserves an IP address for a client after receiving its discover message. Complete proofs appear in the full paper.

Theorem 5.8. *Let $\xi_1 = \theta$, and $\xi_2 = \mu$. Then any execution of $\mathcal{C}$ satisfies the request liveness property in Definition 3.7.*

Proof. Despite the complicated statements of the request and renew liveness properties, it is in fact straightforward to show that $\mathcal{C}$ satisfies them. This is because the properties are basically a list of all the problems which might occur to prevent liveness. Thus, the liveness proof consists of showing that when all such problems are ruled out, $\mathcal{C}$ is live. □

Theorem 5.9. *Any execution of $\mathcal{C}$ satisfies the renew liveness property in Definition 3.7.*

Combining Theorems 5.7, 5.8 and 5.9, we have shown that $\mathcal{C}$ satisfies all the properties of a DHCPF protocol.

6 Conclusions

In this paper, we presented a formal specification of a fault-tolerant DHCP algorithm for unique IP address assignment. The algorithm is implemented as a composition of two algorithms, modeling dynamic versions of the leader election and shared register problems. This structure facilitated the proof of correctness of the algorithm. Its simplicity also lends well to practical implementations and deployment.

There are several directions for extending our work. While we feel that DHCPF is naturally modeled as a timed mutual exclusion problem, as in our Definition 3.7, there seems to be substantial freedom in choosing the various parameters and assumptions making up this definition. For example, is a failure detector really necessary to implement DHCPF? Do we need stable and timely periods to ensure liveness? If so, can these periods be made smaller than in our definition? Can we find other natural ways to characterize liveness properties, perhaps avoiding the complexity of the request liveness definition? In another vein, having specified DHCPF in a particular way, is the decomposition of the problem into leader election and shared register abstractions the best one? For example, does this decomposition ensure the most independence between the subproblems, so that each problem can be solved in isolation and later composed? Does decomposing DHCPF into subproblems lead to a less efficient, if also less complex solution than solving the problem as a monolithic whole? Some of these questions can be expressed as formal questions of lower bounds. For others, especially the design issues, we currently lack a proper theory to rigorously address them. We hope that our abstract model and analysis contributes to understanding these interesting and important problems.

References

1. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
3. Dolev, D., Shavit, N.: Bounded concurrent time-stamping. SIAM J. Comput. 26(2), 418–455 (1997)
4. Droms, R.: Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard) Updated by RFCs 3396, 4361 (March 1997)
5. Droms, R., Kinnear, K., Stapp, M., et al.: DHCP Failover Protocol (March 2003) `http://www3.ietf.org/proceedings/03mar/I-D/draft-ietf-dhc-failover-12.txt`
6. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: The Theory of Timed I/O Automata. Morgan and Claypool (2005)
7. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)