Automated Formal Verification of the DHCP Failover Protocol Using Timeout Order Abstraction

Shinya Umeno and Nancy Lynch
CSAIL, Massachusetts Institute of Technology, Cambridge MA, USA
{umeno,lynch}@csail.mit.edu

***Abstract*—In this paper, we present automated formal verification of the DHCP Failover protocol. We conduct bounded model-checking for the protocol using *Timeout Order Abstraction* (TO-Abstraction), a technique to abstract a given timed model in a certain sub-class of loosely synchronized real-time distributed systems into an untimed model. A resulting untimed model from TO-abstraction is a finite state machine, and therefore one can verify the model using a conventional model-checker.**

We have verified the protocol by bounded model-checking up to depth 20. We also experimented with "mutating" the original code to examine the efficiency of bug-finding using TO-Abstraction. We used two mutated pieces of the original code. The first one represents a model that uses a stronger failure assumption. The second one represents a model that the protocol implementer has forgot to add a certain check of a received message. We found one counterexample for each of two pieces of mutated code. In particular, the counterexample that was found for the second mutated code had a complex scenario, and we believe that it is considerably difficult to find the counterexample by human or simulations.[1]

1. Introduction

In this paper, we present automated formal verification of the DHCP Failover protocol (DHCP-F, [2]). DHCP-F is an extension of the *Dynamic Host Configuration Protocol (DHCP)*, which is widely deployed for communication devices to automatically obtain an IP address on the Internet. DHCP-F is in the class of loosely synchronized real-time distributed systems (LSRTDS's). In this class of distributed systems, the processes or modules in the system are assumed to have *loose synchronization*, that is, there is an a priori known upper bound ε on the skew between local clocks in processes. Processes communicate *time data* (timing-related information such as time stamps) with each other, and set their *timeouts* using time data. These timeouts are used to constrain processes' behavior in such a way that the processes execute a certain designated action before or after other processes execute another designated action. This type of control of timeout orders is used in DHCP-F to maintain interesting mutual-exclusion and fault-tolerance properties. Namely, the protocol supports no duplication of address assignments even under server failures and recoveries. This property is the one we verify by the presented case study.

This research has been supported by NSF Award 0702670. This paper will appear in Proceedings of 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2010), Oxford, UK, March 22-26 2010. The paper is formatted using a TEX style file for IEEE conferences.

[1]Supplemental files such as SAL code used for the case studies presented in the paper can be obtained from URL cited as [1].

We conduct bounded model-checking for DHCP-F using *Timeout Order Abstraction* (TO-Abstraction, [3]). TO-Abstraction is a technique to systematically abstract an LSRTDS into an untimed model. The subclass of LSRTDS's that we can apply TO-abstraction is defined using a syntax template that represents a restriction to Tempo, the primary modeling language of TIOA [4]. The TIOA framework has been used to model and verify (with hand proofs) several real-time distributed systems and algorithms (for example, [5], [6], [7], [8], [2], [9]). TO-Abstraction enables the user to conduct *time-parametric verification* of a given TIOA model described by the template in the sense that the local clock skew bound ε and the special timing-related constant that we explain later are treated as parameters of the system, and therefore, are not instantiated into concrete values. The untimed model resulting from TO-Abstraction is a finite state machine, and thus one can automatically verify (untimed) temporal properties of the model using a conventional model-checker. The soundness guarantee that we can obtain is that any "untimed" safety property of the untimed model also holds for the original TIOA model. Informally, an untimed safety property is a safety property that refers to non-timing-related part of the protocol. For example, we can specify no two processes enter the critical region at the same time, or the value of a particular counter does not exceed ten in any protocol execution. Basically, any property that we typically use for the ordinary untimed protocol verification can be verified using the untimed model obtained from TO-Abstraction.

Contribution: There are two main contributions in the presented paper. First, as far as we know, this paper presents the first machine-automated formal verification of the DHCP Failover protocol. This protocol is in the class of LSRTDS's, and because of its very general assumption of the local clock value evolutions (as long as the clock values are loosely synchronized with respect to the bound ε, the evolution is described by an arbitrary increasing function), the protocol cannot be modeled using such frameworks as Alur-Dill Timed Automata [10] or Linear Hybrid Automata [11]. Therefore, the protocol cannot directly benefit from the existing verification techniques and tools (for example, UPPAAL [12] and HyTech [13]) developed for these frameworks. The protocol has been studied in [2] in the context of formal verification using manual (hand-written) proofs, but no study on automatic analysis of the protocol has been reported thus far. The authors of [2] proved the correctness of the protocol using elaborate hand-written proofs by induction over system executions. We consider the complexity of formally verifying this protocol is considerably high because of the fact that a complete proof is not included in [2] because a proof consists of proving several non-trivial lemmas needed to prove the final correctness theorem. Our case study provides exhaustive exploration of scenarios of DHCP-F up to the execution length of 20 (20 discrete transitions, including sending and receiving messages, of the system) for the configuration of two clients and two servers.[2] Because this is a bounded guarantee, (with respect to the execution depth and the number of processes), it cannot replace a complete formal proof. However, considering the complexity of lemmas and theorems and their proofs in [2], one cannot be absolutely sure about the correctness of the proofs (unless one conducts a mechanical theorem-proving). In this regard, we believe that our case study helps the community gain more confidence for the correctness of DHCP-F.

Second, we experiment with "mutating" the original code to examine the efficiency of bug-finding using TO-Abstraction. We used two mutated pieces of the original code. The first one represents a model that uses a stronger failure assumption. The second one represents a model that the protocol implementer forgot to add a certain check of a received message. We found one counterexample for each of two pieces of mutated code. In particular, the counterexample that was found for the second mutated code had a complex scenario, and we believe that it is considerably difficult to find the counterexample by human or simulations. This consequence suggests that TO-Abstraction can be used not only for verification, but also for experiments of specification/coding changes in the early-stage development of LSRTDS's.

The rest of the paper is organized as follows. In Section 2, we describe the overview of TO-Abstraction that we use for DHCP-F verification. Section 3 is devoted to explaining how DHCP-F works, and representing the Tempo code that models DHCP-F. Section 4 reports how TO-Abstraction can be applied to DHCP-F and the verification results. We also report experiments for mutated code of the original protocol. In Section 5, we conclude by stating the summary of the paper.

2. Overview of Timeout Order Abstraction

In this section, we describe the overview of TO-Abstraction that we use for DHCP-F verification. We first explain the background of this technique in Section 2.1, next explain the settings and the syntax template used for the technique, and then explain the technique in Section 2.3.

2.1. Background

We have developed Timeout Order Abstraction by analyzing the *DHCP Failover protocol (DHCP-F)*, and trying to find common patterns that satisfy both of the following properties: 1. The set of patterns is general, and other existing real-time protocols may have used it already, or a protocol designer can use it for a future design; and 2. Every protocol described by the set of patterns can be systematically abstracted into an untimed model that can be verified by a conventional model-checker. We have found key building blocks of DHCP-F that other protocols can use under the loose-synchronization assumption. Processes in the protocol use the following two main ways of setting timeouts.

1) The first way of setting a timeout uses a *time nonce*, a value arbitrarily picked by a process.[3] A typical use of a time nonce is described in Protocol 1 in this section.
2) The second way uses a *time stamp*, plus a special fixed constant waiting time u. A time stamp is a value copied from the current value of the local clock of a process. Timeout setting using a time stamp and u can be considered a special form of setting timeout using a time nonce. This special form is used for processes to perform the *time-stamp-estimation* trick, which we explain later in this section.

To provide the reader with the idea of how the above described two ways of setting timeouts can be used, we show the two relatively simple examples. DHCP-F uses similar strategies of the two examples in combination.

We consider the following common setting for the examples. Two processes P_1 and P_2 share a resource, and they must not access the resource at the same time. Their strategy is time-sharing the resource by communicating with each other by sending messages through channels. Two processes' local clocks are loosely synchronized, and the skew between them is strictly less than ε. We assume that the values of their local clocks are monotonically increasing. We assume that for this time-sharing, P_1 first accesses the resource and then P_2 and P_1 alternately accesses the resource in turns. At first, we assume that the channel is stable and thus messages would not be lost and message contains would not become broken.

Protocol 1: P_1 picks the time until which it will use the resource, as a time nonce TN_1, and sends it to P_2. P_1 sets a timeout at TN_1 and starts using the resource. When P_2 receives TN_1, it sets a timeout at $TN_1 + \varepsilon$. The skew bound ε is used to *most conservatively* estimate when P_1 times out, *on P_2's local clock*. When P_2 times out, it is guaranteed that P_1 has already timed out. Therefore, P_2 can *immediately* use the resource. P_2 picks another time nonce TN_2, sets a timeout at TN_2, and sends TN_2 to P_1. Upon receipt of TN_2, P_1 sets its timeout to $TN_2 + \varepsilon$. P_1 and P_2 repeat the same routine forever. □

Next, we consider a different assumption for channels. Now the channels are not stable, and contents of messages sent between processes may become broken. When message contents become broken inside the channel, processes can recognize that the contents are broken (for example, using a check-sum). Under the above assumption, we cannot use Protocol 1 – if information of TN_1 sent from P_1 to P_2 becomes broken, there is no way P_2 can estimate when P_1 finishes its job. One (not so smart) option is going back to "done" message communications.

Using a time stamp with a constant waiting time instead of an arbitrarily time nonce resolves this situation. Suppose processes a priori share the value of the constant waiting time u, and the value of u is fixed.

[2]We conducted a bounded model-checking instead of a full model-checking because the model-checker we used could not treat the untimed DHCP-F model due to the size of the model. We show more details in Section 4.2.

[3]A time nonce in an actual low-level implementation may be computed using, for example, a complex optimization and/or adaptive algorithm or a randomized algorithm. We just assume the most general (least restrictive) assumption: as long as its value is larger than the current clock of the process that picks the time nonce, it is a valid time nonce for our setting.

Protocol 2: P_1 picks a time stamp TS_1, instead of an arbitrary time nonce, and sets its timeout to $TS_1 + u$. Therefore, P_1 uses the resource for u time units (measured on its local clock). P_1 sends $TS_1 + u$ to P_2. If the above message sent to P_2 is not broken, P_2 sets its timeout to $TS_1 + u + \varepsilon$ (the received time data plus ε) as in Protocol 1. If the message is broken, P_2 sets its timeout to the special value $clock_2 + u + 2\varepsilon$, where $clock_2$ is the current value of P_2's local clock. When P_2 times out, it immediately starts using the resource. It also picks a time stamp TS_2, sets its timeout to $TS_2 + u$, and sends $TS_2 + u$ to P_1. When receiving a message from P_2, P_1 sets timeout in a way similar to P_2. P_1 and P_2 repeat the same routine forever. $\square$

Time-Stamp-Estimation Trick: The timeout setting using $clock_2 + u + 2\varepsilon$ in Protocol 2 is the special timeout setting, the *time-stamp-estimation* trick that we have mentioned in the introduction. We explain in the following why this value can be used to estimates conservatively when P_1 times out. We can interpret the value $clock_2 + u + 2\varepsilon$ as $(clock_2 + \varepsilon) + u + \varepsilon$. If $clock_2 + \varepsilon$ is equal to or greater than TS_1, then P_2 indeed succeeds in conservatively estimating P_1's timeout (if the message were not broken, P_2 would have set its timeout to $TS_1 + u + \varepsilon$). We consider in the following the moment that P_2's timeout is set to $clock_2 + u + 2\varepsilon$. From the loose synchronization assumption, the value of $clock_2 + \varepsilon$ is at least as large as the value of $clock_1$, the local clock of P_1. Because P_1's clock value is monotonically increasing, the current value of $clock_1$ is greater than the value of TS_1, which is copied from the past value of $clock_1$. Therefore, $clock_2 + \varepsilon \geq TS_1$, as needed. The key to the above argument was that because of the fact that P_2 received a (broken-content) message from P_1, P_2 was sure that P_1 had already picked time nonce TS_1. The value of $clock_2 + \varepsilon$ overestimates *any time stamp that has been picked thus far in physical time*, and a process can perform this estimation *by looking at just its local clock*.

This time-stamp-estimation trick is used in the DHCP-F model of [2] without particularly mentioning its usefulness, intuition, or subtlety. The Internet-draft version of DHCP-F does not consider this type of subtle argument about loosely synchronized clocks, and thus how long a leading server has to wait to conservatively estimate time stamps other servers have picked is not clearly stated. We contacted the first author of [2], and were informed that this subtle special usage, which we will call the time-stamp-estimation trick, was proposed by him, discussed by the authors of [2], and then adopted in the model of DHCP-F used in [2]. TO-Abstraction can treat the time-stamp-estimation trick by using a special form of abstraction for it. We will explain more details in Section 2.3.

2.2. Setting and Template

TO-Abstraction assumes that the system is executed under the loose-synchronization assumption and each process in the system is described by a syntax template that describes the restrictions to more general Tempo guarded-command-style language (the primary modeling language for TIOA, [4]). The basic idea of the template is to restrict use of time data and timeouts to a special form for which TO-abstraction can be applied. There is no restriction imposed for "untimed" part of the language. In this section, we present more details on this assumption and the template.

Loose-Synchronization Assumption: We assume that processes are *loosely synchronized*: for any pair (P_i, P_j) of processes, the deviation between the values of their local clocks, $clock_i$ and $clock_j$, respectively, are bounded by an a priori known amount ε, as shown in the following inequality: $|clock_i - clock_j| < \varepsilon$. We assume that the ε bound is known to every process as its parameter. We also assume that a constant positive real value u is known to every process as its parameter.

Communication Interface: The template needs the following forms for the signatures (interfaces) of actions for process P_i. The system has output and input actions for message communication between processes, such as 'send' and 'receive'. A 'send' action has the following signature: $\mathrm{send}_i(j\text{:ProcessID}, M\text{:UntimedMessage}, \chi\text{:NonNegReal}, \kappa\text{:InteractionInst})$. This action represents that P_i sends real-valued "time data" χ and "untimed data" M to process P_j. We will explain later in this section the form of time data allowed in the template. Untimed data is an arbitrary value in a bounded domain. An *interaction instance*, κ, is a special identifier of process interactions to distinguish different set of interactions. A 'receive' action has the same type of interface as a 'send', in order to match communications between processes. A 'broadcast' action, bcast_i, has a similar interface as send_i, but instead of a process ID j, it has a set of process IDs J that represents a subset of processes to which the broadcast is performed. A process also has internal actions which are performed without communicating outside of the world.

Timeouts : A timeout of P_i is modeled as an output action, $timeout_i^k$, and has a very specific form for its precondition (the transition guard) so that the timeout happens at the time the local clock of P_i hits a specified timeout time.

State Variables of Processes: State variables in a TIOA that represents process P_i are split into the six groups shown in Table I. This explicit split of variables is the core of why we can apply TO-abstraction to the template.

Group Name	Variables	Types of Variables
Timed variables	$\{x_i^k\}_{k=1}^{\ell_i}$	NonNegReal
Timeout variables	$\{t_i^k\}_{k=1}^{r_i}$	NonNegReal
Timeout setting Booleans	$\{timeout\_is\_set_i^k\}_{k=1}^{r_i}$	Boolean
Interaction instance variables	$\{w_i^k\}_{k=1}^{q_i}$	Integer
Untimed variables	$\{v_i^k\}_{k=1}^{m_i}$	BoundedDomain$_i^k$
Local clock variable	$clock_i$	NonNegReal

TABLE I
STATE VARIABLES OF PROCESS P_i

The first group consists of timed variables. These variables can store real-valued time data such as time nonces and time stamps (plus u), or the result of a 'max' operation of multiple pieces of time data. We will see the form of value assignments that can be used for timed variable later in this section. The second group consists of timeout variables. These variables are special variables to set a time when a timeout action must be performed. They can store time data, possibly plus ε (the assumed upper bound on the skew), and the result of the time-stamp-estimation trick. The third group consists of timeout setting Booleans. These are Boolean variables that are used to indicate whether or not the timeout for $timeout_i^k$ is set. The fourth group consists of interaction instance variables. These variables are used to store existing interaction instances received from process communications. The fifth group consists of untimed variables. These variables can store any bounded-size information other than time data. An untimed variable can be, for example, a Boolean, a (bounded-size) counter, or a finite set of process IDs to aggregate the information about which processes have responded to P_i's message. The last group consists of just one variable, $clock_i$, which represents the local clock of process P_i. The value of this variable evolves over real time, and the evolution is an arbitrary increasing function that satisfies the loose-synchronization assumption.

Timed Expression Template: Due to space limitation, we only show the most important part of the template in this paper. The templates for $TimeDataExpression$ and $TimeoutExpression$ shown in TimedExpression are the most important templates in the entire template set. $TimeDataExpression$ is used to assign a timed variable x_i^k by an assignment $x_i^k := TimeDataExpression$, and $TimeoutExpression$ is used to assign a timeout variable t_i^k by an assignment $t_i^k := TimeoutExpression$. The templates for these two expressions include a time nonce picking (new_time_nonce(), which computes a value greater than $clock_i$, representing a future time), computation of a time stamp plus the constant u ($clock_i + u$), a 'max' operation to conservatively update the timeout estimate (we will see how 'max' is used in DHCP-F in Section 3), using time data χ received from another process, timeout setting using ε, and the time-stamp-estimation trick ($clock_i + u + 2\varepsilon$). These expressions are used in templates for processes' transitions explained later in this section.

$$\begin{aligned}
TimeDataExpression ::= {} & new\_time\_nonce() \mid clock_i + u \mid \\
& x_i^k \mid \chi \mid \\
& \max(TimeDataExpression, \\
& \quad TimeDataExpression) \\
TimeoutExpression ::= {} & TimeDataExpression \mid \\
& TimeDataExpression + \varepsilon \mid \\
& clock_i + u + 2\varepsilon \mid \\
& \max(clock_i + u + 2\varepsilon, \\
& \quad TimeDataExpression + \varepsilon)
\end{aligned} \tag{TimedExpression}$$

Because of the restricted use of time data described by the above templates, we can symbolically represent time nonces and time stamps using symbolic labels of them, as we will explain in Section 2.3.

2.3. Timeout Order Abstraction

In this section, we briefly describe how Timeout Order Abstraction (TO-Abstraction) works for the template explained in Section 2.2. We first explain the intuition behind TO-Abstraction, and then show more details of techniques used.

Intuition behind TO-Abstraction: Our goal for TO-abstraction is to obtain an untimed model that conservatively abstracts the original one. In order to obtain an untimed model, we remove the local clocks of processes, and maintain just the right amount of information that can retrieve correct timeout orders in the original timed model. The key idea is to *not* focus on real values of time nonces and time stamps, but instead use identifiers (IDs) of them (labels in other words). Identifiers are integers that distinguish different time nonces and stamps. We explain in more detail how IDs can be used using toy examples described earlier in this section.

In Protocol 1, time nonces are used to synchronize processes behavior. Processes control the ordering of timeouts using the relative difference of their timeout times measured on their local clocks: one process P_i sets a timeout to a time nonce TN_k, and the other P_j sets it to $TN_k + \varepsilon$. Information that is sufficient to retrieve the fact that P_j times out after P_i has done so in any possible scenario, is that both processes use the same time nonce TN_k, and P_i's timeout is set by adding the "slack" ε (the assumed upper bound on the skew) to the time nonce, but P_i's timeout is set solely by the time nonce. To keep track of the above information, we can use the following abstraction: Picks of real-valued time nonces are replaced by picks of symbolic IDs of them. The k-th picked time nonce TN_k in an execution of the original system is represented by ID (integer) k. The timeout time for P_1 set to TN_k is symbolically represented by a pair $(k, false)$, and the timeout time for P_2 set to $TN_k + \varepsilon$ is symbolically represented by a pair $(k, true)$ – the Boolean values represent whether the slack ε is added or not. Then, we remove the local clocks of both processes. After this abstraction, even though we cannot access to the information of local clock values (since they are removed), we can carefully choose which timeouts *must not occur* at the current state of the untimed model, by looking at the symbolic representations: If P_i's timeout is set to $(k, false)$ and P_j's timeout is set to $(k, true)$, P_j's timeout must not be performed, considering timeout orders that can possibly appear in the original timed model. The above discussion is the basic idea of time data abstraction and timeout order constraining, two of the three techniques that we use in combination for TO-abstraction. We can apply the above idea of maintaining information that is sufficient to retrieve correct timeout orders to Protocol 2 as well. In this case, we use IDs of time stamps. Since processes always add u to time stamps, we can just keep track of the ID of a time stamp when the value computed from a time stamp plus u (in the form of $TS_k + u$) is communicated.

In some protocols including DHCP-F, processes use both time nonces and time stamps, and combine them using 'max' operations. For example, a process can combine two time nonces and three time stamps. In such cases, we can extend the same idea by using two sets of integers (IDs) to represent one time data: $(\{1, 2\}, \{4\})$ represents $\max(TN_1, TN_2, TS_4 + u)$, and $(\{3\}, \{2\}, true)$ represents a timeout set at $\max(TN_3, TS_2 + u) + \varepsilon$.

The time-stamp-estimation trick using $clock_i + u + 2\varepsilon$ is abstracted as $(\{\}, picked\_ts\_id\_set, true)$, where $picked\_ts\_id\_set$ is the set of time stamp IDs that have been picked thus far in the current execution. $(\{\}, picked\_ts\_id\_set, true)$ represents $\max(picked\_ts\_set) + u + \varepsilon$ in the original model, where $picked\_ts\_set$ is the set of time stamps picked thus far. This reflects the discussion for the time-stamp-estimation trick earlier in this section that $clock_i + u + 2\varepsilon$ can be interpreted as $(clock_i + \varepsilon) + u + \varepsilon$, and $clock_i + \varepsilon$ over-approximates every time stamp that has been picked in physical time.

Three Techniques Used for TO-Abstraction: Now we explain techniques used for TO-Abstraction in more details. Due to space limitation, we explain only the part of the techniques that are relevant to the DHCP-F case study.

We use three sub-techniques in combination for timeout oder abstraction of a TIOA model described by the syntax template for TO-Abstraction. The three techniques are: 1. Time data abstraction, 2. Timeout order constraining, and 3. Time data reuse and compression. The first two techniques are used to abstract the underlying real-time system into an untimed, but infinite-state model. The third technique is used to represent the infinite state space of the untimed abstraction using a finite one.

[Time Data Abstraction]: First, *time data abstraction* abstracts away local clocks from a given TIOA model, and abstracts real-valued time data in the system using symbolic representations, as discussed earlier in this section. The time-stamp-estimation trick using $clock_i + u + 2\varepsilon$ is abstractly represented by $(\{\}, picked\_ts\_id\_set, true)$. In addition, all boolean conditions in the original model that require a local clock value to determine their truth-values are conservatively approximated. Processes may use an *expiration check* of the form $clock_i < \chi$, by which it examines whether a received time data χ is not "expired", that is, χ represents a future time. These expiration checks are approximated using a special list, called the expired time-data list, which stores time nonce and time stamp IDs picked thus far that are "globally expired" – expired on *all* local clocks. Whether a time nonce/stamp ID ID_k is strongly expired is deduced when a timeout is performed at $TD + \varepsilon$, and symbolic time data TD contains ID_k, and in such a case, ID_k is included in the expired time-data list. Using the expired list, we can infer the truth value of $clock_i > x$ in the following case: if all IDs in x are strongly expired, then $clock_i > x$ must be true. If we cannot infer the truth value of the expiration check, then the abstraction uses a non-deterministic choice. Time Data Abstraction is formally defined in [3] as a function from Tempo code described using the template to another Tempo code that does not use any timing information.

[Timeout Order Constraining]: The untimed model after time data abstraction looses the control over timeout orders in the original timed model because the abstraction removes local clocks of processes. Therefore, we need to put the correct timeout orders back into the untimed model by constraining timeout orders using the information from symbolically represented timeout time, as we briefly describe earlier in this section. Considering the loose synchronization assumption, if a timeout action $timeout^i_k$ is set to a time larger than the time for another timeout $timeout^j_\ell$ by more than ε, then $timeout^i_k$ occurs after $timeout^j_\ell$ has done so. By rephrasing the above statement in symbolic representations, we obtain the following. Timeout action $timeout^i_k$ of a process i is disabled when the following condition holds: $\exists t^j_\ell \in Timeout\_variables_j : (j \neq i) \wedge (\mathsf{tn\_set}(t^j_\ell) \subseteq \mathsf{tn\_set}(t^i_k)) \wedge (\mathsf{ts\_set}(t^j_\ell) \subseteq \mathsf{ts\_set}(t^i_k)) \wedge \neg\mathsf{slack\_added}(t^j_\ell) \wedge \mathsf{slack\_added}(t^i_k)$, where $\mathsf{tn\_set}(t^j_\ell)$ and $\mathsf{ts\_set}(t^j_\ell)$ represents the time nonce ID set and time stamp ID set, respectively, that symbolically represents the combined time nonces and stamps using 'max', and $\mathsf{slack\_added}(t^j_\ell)$ represents whether or not the Boolean flag that represents the slack ε is added to the underlying timeout, is set to true.

[Time Data Reuse and Compression]: An execution of the untimed model may require an infinite number of new time nonce and/or time stamp IDs because the length of a model execution is typically infinite. Our basic strategy is to *reuse* a time stamp or a time nonce that is once taken by some process but is no longer used anywhere in the model. We use a more elaborate *Time Data Compression* technique for DHCP-F. Time Data Compression is used to express multiple 'max'ed time nonces (or stamps) by one symbolic ID. For example, when processes in the system use either $\max(TN_1, TN_2)$ or TN_3 in the current state of the system, we can consider $\max(TN_1, TN_2)$ as one cluster of time nonces, and therefore represent it by one ID. More details of how we can find such "compress-able" time nonce or stamp IDs and how a compression is actually performed is described in [3]. The above described reuse and compression techniques are not technically needed to conduct a bounded model-checking (not a full model-checking), since processes can use only a finite number of IDs in a bounded depth. However the techniques can contribute to reduce the ID space needed for a "valid" bounded model-checking – for model-checking, we first bound the size of the ID space, and conduct a model-checking. If we find that the ID space is actually too small and therefore a fresh ID is not available in some execution (within a certain depth for a bounded model-checking), then we increase the size of the ID space, and run the model-checker again. We used this "bound-and-supersize" strategy for the presented case study, in order to find the most compact sufficiently large ID space.

A soundness guarantee that we can obtain for TO-Abstraction is that for any execution of the resulting untimed model, there is a corresponding execution of the original automaton such that the values of all untimed variables are the same in any pair of two states that respectively appear after the same number of discrete actions in the two executions. This soundness guarantee is proved in [3] using a simulation relation proof technique.

3. DHCP Failover Protocol

The *DHCP Failover protocol (DHCP-F)* is an extension of the *Dynamic Host Configuration Protocol (DHCP)*, which is widely deployed for communication devices to automatically obtain an IP address on the Internet. Upon a request from a client, the DHCP server automatically assigns an IP address to the client. The server gives an assignment in the form of a "lease": an IP address assignment from the server is associated with its expiration time until which a client is allowed to use the address.

DHCP-F supplements the ordinary DHCP with stronger fault tolerance using multiple backup servers – when the main server encounters a failure and becomes down, one of the backup servers takes over the main server's job. The main difficulty of using such backup servers is to maintain the consistent view of the lease periods of IP addresses across the main server and all backup servers. Most standard database consistency techniques cannot be used for DHCP-F because they are too slow for this application. For this reason, DHCP-F uses the combination of the two stage assignment scheme.

The first assignment is a short assignment that uses a time stamp plus the constant waiting time u. This assignment is fast, requiring no acknowledgment communication, but limits how long the address can be used by one client. The second assignment is slower, and used by clients to *renew* its lease. It requires explicit acknowledgment from all backup servers, but a client can use one address indefinitely by renewing the lease.

DHCP-F is described in an Internet Draft that is over 130 pages long. This length is primarily due to the need to deal with many types of concurrent server failures.

In [2], Fan et al. analyzed DHCP-F using the TIOA framework and hand-written proofs of correctness. They *decomposed* the DHCP-F protocol into two sub-protocols: the *leader elector* and the *address assigner*. The leader elector elects one of the backup servers as the next main server when the current main server encounters a failure. The address assigner assigns an IP address to a client upon the request from it. We will explain how this assignment process works in Section 3.1.

The leader elector uses a *failure detector*. When the main server encounters a failure, the failure detector notifies the leader electors (implemented in a distributed fashion in each backup server) of this information. The safety property of the leader elector (only one server is elected as the main server at any time) is guaranteed by waiting long enough time since the elector recovers from a failure. The leader elector sub-protocol and its analysis of the correctness is relatively simple compared to the address assigner, which conducts the main task of the DHCP-F protocol. Therefore, in this case study, we will focus mainly on the address assigner protocol of DHCP-F, and we model the leader elector as a simple helper module that notifies one backup server of the fact that the server is elected as the new main server.

The main safety property of DHCP-F that we verify in this paper is the no-duplicated-address-assignment property – one specific address is assigned to at most one process. In [2], the authors also verified interesting timeliness properties of the protocol as well as the no-duplicated-address-assignment property. Verification of timeliness properties using a machine-automated method is a challenging future study.

Because we have to use a model-checker to verify the untimed abstraction, we need to fix the configuration of the system. We look at the minimum interesting configuration, which consists of one main server, one backup server, and two clients (for the possibility of duplicated assignments).

An actual implementation of the DHCP-F server handles multiple IP addresses concurrently. In this case study, the model handles just one IP address. This is because this concurrent treatment of the IP addresses in one server can be viewed as running multiple threads (or processes) each of which treats exactly one IP address, and these threads do not affect each other. Therefore, to verify the no-duplicated-address-assignment property of DHCP-F, it is sufficient to focus on one thread that handles one IP address.

3.1. TIOA code of DHCP-F

Due to space limitation, we only present the transition definition of the server automaton (Fig. 1) and the client automaton (Fig. 2). The complete code appears in [3]. The fail input comes in from the environment at any time. The "leader-elector" helper module outputs the lead action to the alive server with the minimum ID when the module observes a failure in the current execution. In the code in Fig. 1, we assume that when a sever encounters a failure, all variables except for potlease are reset.

In this code, lease_expired$_i$ is the only timeout action in the server and the client. acklease, potlease, and bcast_value are the tree timed variables x_i^1, x_i^2, and x_i^3. timeout represents the only timeout variable t_i^1.

The protocol works as follows in the nominal operation. First, a client broadcasts a request with some time nonce τ, representing the time until which a client wants an IP address. It picks a new interaction instance κ for this session, and sends τ with a message type 'Request' and κ. When the main server receives the request, it sends back an acknowledgment with the lease offer for a *short lease*, using $clock_i + u$. It also updates its current interaction instance to κ. Then the server broadcasts the most conservative potential lease time (a 'max' of τ and $clock_i + u$) with κ to all backup servers. When a client receives a lease offer, and if the lease time has not expired yet, it accepts the offer and uses the IP address until the specified lease time. When a backup server receives a potential lease time from the main server, it updates its own estimate of the most conservative potential lease time stored in potlease, and sends back an acknowledgment to the main server with κ. The main server collects acknowledgements by accepting only acknowledgements for the current session (judged by the interaction instance of the acknowledgements). When the main server has collected acknowledgments from all backup servers, it updates the value of acklease, the variable used to respond a lease-renew request from a client. This implies that the time nonce τ first sent by a client to the main server is used *only after* the main server collects acknowledgements from the backup servers. When the main server receives a renew request with another time nonce τ_2 from the current client after the above described "collection" period, it offers the lease until $\max(\tau, TS_1 + u, clock_i + u)$, where TS_1 is the time stamp taken when the main server first responds to the current client. Then the main server enters the "collection" period for the new time nonce τ_2 ('max'ed with previously accumulated time data in potlease). Similarly, for each following renew message, the client is offered the time nonce for the one-session-previous renew (or request) message, maxed with a new $clock_i + u$ of the main server and accumulated other time data in acklease.

When the main server encounters a failure, the leader-elector detects it, and assigns a new main server. Recall that a lease time (a time nonce) requested by the client is always used only after collecting acknowledgements from backup servers. When a backup server inputs lead, it sets its timeout to 'max' of the potential lease time and '$clock_i + u + 2\varepsilon$' (the time-stamp-estimation trick), so that it times out after all time nonces stored in the potential lease time, potlease, and all existing time stamps that a client could be using has expired. This intricate use of the time-stamp-estimation trick enables the main server to offer a short lease of $clock_i + u$, *without collecting acknowledgements from backup servers*.

As described above, usage of maximum operations and the time-stamp-estimation trick used in DHCP-F is very subtle,

```
input receive_j($i$, 'Request', $\tau$, $\kappa$)
 eff  if leading$\wedge$current_user = $\bot$
     $\wedge$ pc = waiting then
       current_user := $i$;
       current_interact_inst := $\kappa$;
       acklease := clock$_j$ + $u$;
       potlease := acklease;
       bcast_value :=
             max($\tau$, acklease);
       timeout_time := potlease + $\varepsilon$;
       timeout_is_set := true;
       pc := ack_to_user;   fi

output send_j($i$, 'Ack', $\chi$, $\kappa$)
 pre  $\chi$ = acklease $\wedge$
      pc = ack_to_user $\wedge$
      $i$ = current_user $\wedge$
      $\kappa$ = current_interact_inst
 eff  pc := bcast_potlease;

output bcast_j(AllServers,
               'PotleaseWrite', $\omega$, $\kappa$)
 pre  pc = bcast_potlease $\wedge$
       $\omega$ = bcast_value $\wedge$
       $\kappa$ = current_interact_inst
 eff  pc: = waiting;
  for j':Server_ID:
    potlease_ack_rcvd[j'] := (j' = j);

input receive_j($j'$,
               'PotleaseWrite', $\omega$, $\kappa$)
 eff  if $\neg$ leading $\wedge$ pc $\neq$ down
      then
       bcast_value := $\omega$
       potlease := max(potlease, $\omega$)
       potlease_ack_leader := $j'$
       pc := send_potlease_ack   fi

output send_j($j'$,
            'PotleaseWriteAck', $\omega$, $\kappa$)
 pre  $\omega$ = bcast_value $\wedge$
      $\kappa$ = current_interact_inst $\wedge$
      pc = send_potlease_ack
 eff  pc := waiting;

input receive_j($j'$, 'PotleaseWriteAck',
            $\omega$, $\kappa$)
 eff  if $\kappa$=current_interaction_inst$\wedge$
      leading $\wedge$ pc $\neq$ down then
       potlease_ack_rcvd[j'] := true
       if $\forall$ (j:Server_ID):
          (potlease_ack_rcvd[i]) then
         acklease :=
          max(acklease, $\omega$)    fi fi

input receive_j($i$, 'Renew', $\tau$, $\kappa$)
 eff  if leading $\wedge$ current_user = $i$
    $\wedge$ pc $\neq$ down then
      current_interact_inst := $\kappa$;
      acklease :=
        max(acklease, clock$_j$ + $u$);
      potlease :=
        max(potlease, acklease);
      bcast_value :=
        max($\tau$, acklease);
      timeout_time := potlease + $\varepsilon$;
      timeout_is_set := true;
           pc := ack_to_user;   fi

output lease_expired_j
 pre  timeout_time = clock$_j$ $\wedge$
      timeout_is_set
 eff  timeout_time := 0;
  timeout_is_set := false;
  current_user := $\bot$;
  pc := waiting;

input fail_j
 eff  pc := down
  timeout_time := 0
  timeout_is_set := false
  leading := false
  current_user := $\bot$
  current_interact_inst := $\bot$
  acklease := 0
  bcast_value := 0
  for j':Server_ID:
    potlease_ack_rcvd[j'] := false

input recover_j
 eff  pc := waiting;

input lead_j
 eff  leading := true;
  timeout_time :=
    max(clock$_j$ + $u$ + $2\varepsilon$,
         potlease + $\varepsilon$);
  timeout_is_set := true;
  pc := preparing;
```

Fig. 1. Transitions of the server automaton of DHCP-F

```
output bcast_i(AllServers,
        'Request', $\tau$, $\kappa$)
 pre  $\tau$ = new_time_nonce() $\wedge$
    $\kappa$ = new_interact_inst() $\wedge$
     pc = requesting

input receive_i($j$, 'Ack', $\chi$, $\kappa$)
 eff  if $\chi$ > clock$_i$ $\wedge$
       $\kappa$ = current_interact_inst
     then
      timeout := $\chi$;
      timeout_is_set := true;
      pc := leasing;   fi

output send_i($j$, 'Renew', $\tau$, $\kappa$)
 pre  $\tau$ = new_time_nonce() $\wedge$
    $j$ = current_server $\wedge$
    $\kappa$ = new_interact_inst() $\wedge$
    pc = leasing

output lease_expired_j
 pre  timeout = clock$_i$ $\wedge$
      timeout_is_set
 eff  timeout := 0;
  timeout_is_set := false;
  pc := idle;
```

Fig. 2. Transitions of the client automaton of DHCP-F

and therefore it is hard to manually analyze whether or not the timeout is set as intended. Thus, we benefit from the power of automated analysis given by TO-abstraction.

4. Model-Checking DHCP-F Using TO-Abstraction

In this section, we report the automated formal verification of DHCP-F using TO-Abstraction. In Section 4.1, we show the code of DHCP-F after TO-Abstraction. Section 4.2 shows the verification results. We also experiment with "mutated" version of the original code to examine the efficiency of bug-finding using TO-Abstraction.

4.1. Untimed Abstraction of DHCP-F

In this section, we present the untimed abstraction of DHCP-F. We show in Figures 3 and 4 the untimed version of code described in Figures 1 and 2, respectively. The yellow-colored parts of the code are the parts changed by the abstraction.

We use the following functions in the untimed version:

1) symb_max: used to represent a symbolic version of 'max' operations. This is implemented by element-wise union to the pair of time-nonce and time stamp ID sets.
2) set_timeout: used to represent the symbolic timeout value using symbolic time data and the boolean for slack-added or not.
3) ts_set: used to represent the time stamp ID set in the symbolic time data. tn_set is similarly defined.
4) expired: used to represent the expiration implication from the expired time-data list that we described in Section 2.3.

As we described in 2, all timed and timeout variables now use symbolic representations with time stamp and time nonce IDs. The time-stamp-estimation trick is approximated using picked_ts_ID_set, the time stamp IDs picked thus far in the current protocol execution. The expired time-data list consists of two sublists, expired_ts_ID_list and expired_tn_ID_list, and updated when timeout with a slack ε is performed. Boolean variable rcvd_time_expired is used to determine the truth-value of the expiration check of the form of $clock_i < \chi$, using the expired time-data list.

4.2. Verification Results

We used the SAL model-checker [14] developed by SRI International to conduct verification of the untimed abstraction of DHCP-F. For this case study, we manually abstracted the timed model into the untimed model described in the SAL model. We tried to define every data structure in SAL as general as we can, so that the structure can be reused for other case studies. The actual code can be found in [1]. We are currently implementing an automatic abstraction (code-conversion) tool for TO-abstraction.

The full model-checking using BDD was not feasible for this case study arguably due to the complexity of the protocol. (We encountered the out-of-memory error when SAL was computing the transition function, that is, even before verification.) The number of time stamps and time nonces were increased when the time-stamp/time-nonce unavailability error has occurred. All experiments are conducted using a Linux machine with an Intel Core^{TM} 2 Quad at 2.66 GHz and 4GB memory.

We used the minimum interesting configuration, which consists of one main server, one backup server, and two clients (for a possible duplicated assignment). We used a feature

```
input receive$_j$($i$, ‘Request’, $\tau$, $\kappa$)
  eff  if leading$\wedge$current_user = $\bot$
      $\wedge$ pc = waiting then
        current_user := $i$;
        current_interact_inst := $\kappa$;
        acklease :=
        new_ts_ID(picked_ts_ID_set);
        potlease := acklease;
        bcast_value :=
            symb_max($\tau$, acklease);
        timeout_time :=
         set_timeout(potlease, true);
        timeout_is_set := true;
        pc := ack_to_user;   fi;
        picked_ts_ID_set :=
        one_ts_ID_picked(
              picked_ts_ID_set)

output send$_j$($i$, ‘Ack’, $\chi$, $\kappa$)
  pre  $\chi$ = acklease $\wedge$
    pc = ack_to_user $\wedge$
    $i$ = current_user $\wedge$
    $\kappa$ = current_interact_inst
  eff  pc := bcast_potlease;

output bcast$_j$(AllServers,
          ‘PotleaseWrite’, $\omega$, $\kappa$)
  pre  pc = bcast_potlease $\wedge$
     $\omega$ = bcast_value $\wedge$
     $\kappa$ = current_interact_inst
  eff  pc: = waiting;
   for j’:Server_ID:
    potlease_ack_rcvd[j’] := (j’ = j);

input receive$_j$($j'$,
          ‘PotleaseWrite’, $\omega$, $\kappa$)
  eff  if $\neg$ leading $\wedge$ pc $\neq$ down
      then
      bcast_value := $\omega$
      potlease :=
        symb_max(potlease, $\omega$)
      potlease_ack_leader := $j'$
      pc := send_potlease_ack   fi

output send$_j$($j'$,
        ‘PotleaseWriteAck’, $\omega$, $\kappa$)
  pre  $\omega$ = bcast_value $\wedge$
    $\kappa$ = current_interact_inst $\wedge$
    pc = send_potlease_ack
  eff  pc := waiting;

input receive$_j$($j'$, ‘PotleaseWriteAck’,
        $\omega$, $\kappa$)
  eff  if $\kappa$=current_interaction_inst$\wedge$
      leading $\wedge$ pc $\neq$ down then
      potlease_ack_rcvd[j’] := true
      if $\forall$ (j:Server_ID):
        (potlease_ack_rcvd[i]) then
       acklease :=
       symb_max(acklease, $\omega$)   fi fi

input receive$_j$($i$, ‘Renew’, $\tau$, $\kappa$)
  eff  if leading $\wedge$ current_user = $i$
   $\wedge$ pc $\neq$ down then
     current_interact_inst := $\kappa$;
     acklease :=
      symb_max(acklease,
         new_time_stamp_ID(
            picked_ts_ID_set));
     potlease :=
      symb_max(potlease,
          acklease);
     bcast_value :=
      symb_max($\tau$, acklease);
     timeout_time := potlease + $\varepsilon$;
     timeout_is_set := true;
     pc := ack_to_user;
     one_ts_ID_picked(
            picked_ts_ID_set); fi

output lease_expired$_j$
  pre  timeout_is_set
  eff  if slack_added(timeout_time)
     then
     expired_ts_ID_list :=
       expired_ts_ID_list $\cup$
       ts_set(timeout_time);
     expired_tn_ID_list :=
       expired_tn_ID_list $\cup$
       tn_set(timeout_time); fi
   timeout_time := ({},{},false);
   timeout_is_set := false;
   current_user := $\bot$;
   pc := waiting;

input fail$_j$
  eff  pc := down
   timeout_time := ({},{},false);
   timeout_is_set := false
   leading := false
   current_user := $\bot$
   current_interact_inst := $\bot$
   acklease := ({},{})
   bcast_value := ({},{})
   for j’:Server_ID:
     potlease_ack_rcvd[j’] := false

input recover$_j$
  eff  pc := waiting;

input lead$_j$
  eff  leading := true;
   timeout_time :=
     set_timeout(symb_max(
        ({picked_ts_ID_set},{}),
                potlease),
                true);
   timeout_is_set := true;
   pc := preparing;
```

Fig. 3. Transitions of the server automaton of DHCP-F, untimed version

```
output bcast$_i$(AllServers,
      ‘Request’, $\tau$, $\kappa$)
  pre  $\tau$ = new_time_nonce_ID(
         picked_tn_ID_set)) $\wedge$
     $\kappa$ = new_interact_inst() $\wedge$
     pc = requesting
  eff picked_ts_ID_set :=
    one_ts_ID_picked(
         picked_ts_ID_set)

input receive$_i$($j$, ‘Ack’, $\chi$, $\kappa$)
  eff  rcvd_time_expired :=
     if expired($\chi$,
       expired_ts_ID_list,
       expired_tn_ID_list)
     then true
     else choose(true, false)
    if $\neg$ rcvd_time_expired $\wedge$
      $\kappa$ = current_interact_inst
    then
    timeout:=set_timeout($\chi$, false);
     timeout_is_set := true;
     pc := leasing;   fi

output send$_i$($j$, ‘Renew’, $\tau$, $\kappa$)
  pre  $\tau$ = new_time_nonce_ID(
         picked_tn_ID_set)) $\wedge$
    $j$ = current_server $\wedge$
    $\kappa$ = new_interact_inst() $\wedge$
    pc = leasing
  eff picked_ts_ID_set :=
    one_ts_ID_picked(
         picked_ts_ID_set)

output lease_expired$_j$
 pre  timeout_is_set
 eff  if slack_added(timeout_time)
   then
   expired_ts_ID_list :=
     expired_ts_ID_list $\cup$
     ts_set(timeout_time);
   expired_tn_ID_list :=
     expired_tn_ID_list $\cup$
     tn_set(timeout_time); fi
  timeout_time := ({},{},false);
   timeout_is_set := false;
   pc := idle;
```

Fig. 4. Transitions of the client automaton of DHCP-F, untimed version

of SAL that iteratively increases the depth for the bounded model-checking up to a specified depth bound to search for a counterexample. We succeeded in verifying the protocol up to depth 20 under the failure assumption that when a server encounters a failure all variables *except for potlease* are reset. The total verification time was 350743.7 seconds (97.43 hours). Table II shows individual times for the bounded model-checking of different depths and accumulated times that took up to each of the depths (we only show depth 15 and higher). We consider that the depth of 20 covers several subtle scenarios and is arguably sufficiently deep for finding counterexamples that could be potentially encountered in a real implementation. This is because, as we show later in this section, when we mutate one check condition in one transition of the original code, we can find at depth 17 a counterexample that has a fairly complex scenario.

Depth	Individual time (hours)	Accumulated time (hours)
15	2.70	5.54
16	5.00	10.54
17	6.45	16.99
18	14.72	31.71
19	23.84	55.55
20	41.88	97.43

TABLE II
BOUNDED MODEL-CHECKING TIMES

First Mutation (Different Failure Assumption): Next, we experiment with another failure assumption that when a server encounters a failure, all variables are reset. Under this assumption, we have found a counterexample at depth 17.[4]

[4]This counterexample is found with the configuration that the number of server is one, and the number of clients are two. We are currently running experiment for the case of two servers and two users. We are expecting to find a similar counterexample at depth 20. The illustrated counterexample is for two servers.

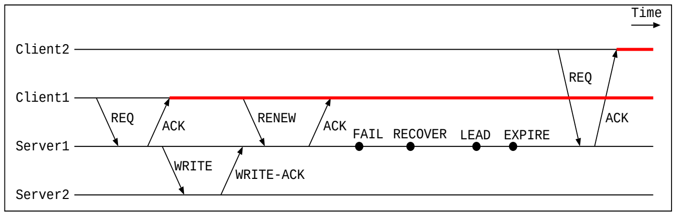

Fig. 5. Duplicated lease scenario when all variables are reset by a failure

This counterexample is illustrated in Fig. 5.[5] This scenario is actually realizable in the original timed system as well, as described in the following:

1) Client-1 sends a lease request that contains a large time nonce τ.
2) Server 1 sends a Write message to Server 2, and Server 2 sends back a Write-Ack message; and thus Server-1's acklease is updated (now the maximum contains τ).
3) Client-1 sends a Renew message, and Server 1 receives it. acklease is updated to max(acklease, ts + u), which is no smaller than τ. And then, potlease is updated to max(potlease, acklease).
4) A lease offer until max(acklease, ts + u) is sent to Client 1. Client 1 receives it, and updates its lease timeout to max(acklease, ts + u).
5) Server 1 encounters a failure, and all variables (especially potlease) are reset. When Server 1 recovers and gets a lead input, it sets the lease timeout to $max(clock_i + u + 2\varepsilon, \text{potlease} + \varepsilon)$, where the value of potlease is 0. Therefore, if τ is larger than the value of $clock_i + u + 2\varepsilon$ at this moment, Server-1 expires before Client-1 expires.
6) Since Server-1 expires, it accepts a request from another client, Client 2, and hence ends up with a duplicated lease.

This counterexample is expected to appear under the assumption of using volatile memory for potlease. This is because potlease stores information of the accumulated information of the potential lease time for the current client. If this information is reset, there is no way the server can retrieve what time nonces the client is now using for its lease time.

Second Mutation (Removing Interaction Instance Check): To examine the effectiveness of bug-finding using TO-abstraction, we experimented with verification of DHCP-F with a slight change. We removed one condition for checking the interaction instance of the received acknowledgement from a backup server to ensure that the received acknowledgement is for the current session. The only change is in the effects of the action receive(j, "WriteAck", ω, κ) of the server automaton. We removed the condition "κ = current_interact_inst" from the if statement. The main server now accepts any "PotleaseWriteAck" acknowledgment as an acknowledgment to the latest "PotleaseWrite" message.

[5]We obtained from the counterexample more information than just communication sequence of processes. For example, how potlease and acklease are changed (using symbolic IDs). From these informations, we constructed the realizable scenario in the original real-time model by considering what kind of assumption for the actual values of time nonces are needed to realize the counterexample execution found in the symbolic world.

After this mutation, we found a counterexample at depth 17 under the configuration of two servers and two users (the run time was 52851.28 seconds $\sim$ 14.7 hours). This counterexample has a complex scenario, and we believe that it is considerably hard to find by human, or simulations with fixed values of ε and u, and random assignments of time nonces.

The counterexample is depicted in Figure 6. A red line in a client's time line expresses that the client is using a lease.

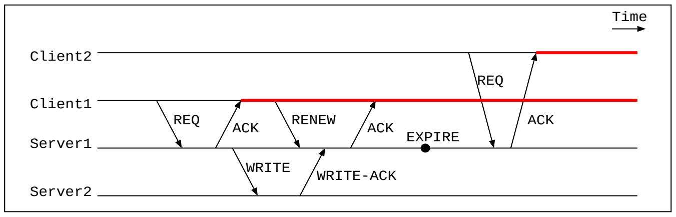

Fig. 6. Duplicated lease scenario for the mutated DHCP-F

We now explain why this counterexample is a real counterexample that could occur in the original real-time mutated protocol as well. Suppose the request lease expire time τ from Client-2 is very large and the (potential) lease time covers the entire time frame depicted in Figure 6.

The key of this counterexample is that the main server (Server-1) receives the "WriteAck" message from the backup server (Server-2) between the receipt of the renew message from a client (Client-1) and the acknowledgment for the renew.

We explain the entire scenario in details in the following.

1) Client-1 sends a lease request that contains a large time nonce τ.
2) Server-1 (the main server) receives the request from Client-1, and replies a *short-lease* acknowledgment containing the offer until $ts_1 + u$, where ts_1 is a time stamp picked by Server-1 at this point. Note that τ is not included in the short lease. The value of bcast_value changes to $\max(\tau, ts_1 + u)$.
3) Client-1 receives the short lease offer and sends a renew message with a new time nonce τ_2.
4) Server-1 broadcasts the "PotleaseWrite" message with $\max(\tau, ts_1 + u)$ (the value of bcast_value) to all backup servers (just one backup in this case).
5) Server-1 receives a renew message from Client-1. Server-1 changes the timeout time (the value of timeout) to $\max(\tau + u, \tau_2 + u) + \varepsilon$, where where ts_2 is a time stamp picked by Server-1 at this point. It also changes the current interaction instance to the one in this renew message.
6) Server-1 then receives "PotleaseWriteAck" from the backup server, Sever-2, *before* acknowledging the renew message from Client-1. The value of current_interact_inst has changed since Server-1 broadcasts the "PotleaseWrite" message. However, the server's program is mutated in such a way that it does not check the interaction instance to decide to accept a received acknowledgment or not. Therefore, Server-1 accepts this "PotleaseWriteAck" from Server-2, and since Server-2 is the only backup server, Server-1 updates the value of acklease used for an acknowledgment

to a renew request from the current client. This update is conducted by calculating the maximum of the current value of acklease and the received timed value ω, which contains the large τ.

7) Client-1 receives the renewing-lease offer from Server-1, and set its lease timeout to the maximum that contains τ.
8) Since τ is very large, the timeout set to Server-1 at time $\max(\tau + u, \tau_2 + u) + \varepsilon$ times out before Client-1's lease expires.
9) Client-2 sends a request to Server-1, and Server-1 offers a lease since it has already timed out.
10) Client-2 receives the lease offer and starts using the IP address.

The above described scenario shows that the usage of the maximum operations used in the original DHCP-F protocol are very intricate: the maximum operations used for potlease, acklease, and bcast_value in the main and backup servers control the timing-related behavior of the servers and clients (by offering a lease time), and how they control this timing-related behavior in a correct fashion is very non-trivial. The atomicity assumption for the send and receive actions also affects the correctness of the protocol: The above scenario would not occur if the main server can receive a renew message and send an acknowledgment for it in an atomic fashion. The atomicity assumption depends on how DHCP-F is implemented. For example, the atomicity granularity we are using in our model is actually realized when broadcasting task and receiving task are operated by two different threads without using lock/unlock. The above discussed facts about this counterexample found for the mutated protocol strongly suggests that we need to use an automated exhaustive analysis technique such as TO-abstraction for this kind of real-time protocols.

5. Conclusion

In this paper, we present a case study on automated formal verification of the DHCP Failover protocol using Timeout Order Abstraction (TO-Abstraction). We successfully verified the protocol using bounded model-checking up to depth 20. We also experiment with two "mutated" versions of the original code. We found a counterexamples for each of the mutated versions, and one of them had a complex scenario of process interleaving that we believe could have not been able to find by human or simulations with fixed system parameters and random assignments of time nonces. This fact infers that TO-Abstraction is helpful not only for verification, but also bug-finding of the protocols. This bug-finding feature is particularly useful when the underlying protocol is still in the development stage, and the designer wants to try slightly different specifications and implementations and the correctness holds for them.

We could not conduct a full model-checking for DHCP-F (even though TO-Abstraction has a potential to conduct it). Therefore we conducted a bounded model-checking for resulting untimed finite state model. This was due to the complexity of DHCP-F. If one aims for only a bounded model-checking from the first point, he/she might be able to directly model the protocol without untiming abstraction, and use satisfiability-modulo-theory (SMT) solver to conduct a bounded model-checking. The approach closest to the above described strategy we can refer to is *Calendar Automata* approach ([15], [16], [17]). It is a interesting future study to examine whether the approach can be applied to DHCP-F, and if so, examine the efficiency compared to the presented case study.

References

[1] S. Umeno and N. Lynch, "Supplemental files for our iceccs-2010 paper," the files can be obtained from `http://people.csail.mit.edu/umeno/dhcpf`.

[2] R. Fan, R. Droms, N. Griffeth, and N. Lynch, "The DHCP failover protocol: A formal perspective," in *27th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems (FORTE 2007)*, ser. Lecture Notes in Computer Science, vol. 4731. Springer, 2007, pp. 208–222.

[3] S. Umeno and N. Lynch, "Timeout order abstraction for formal verification of loosely synchronized real-time distributed systems," Massachusetts Institute of Technology, Tech. Rep., to appear soon. A stable draft version is available from [1].

[4] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2006.

[5] T. Nolte and N. Lynch, "Self-stabilization and virtual node layer emulations," in *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium (SSS 2007)*, ser. Lecture Notes in Computer Science, vol. 4838. Springer, 2007, pp. 394–408.

[6] ——, "A virtual node-based tracking algorithm for mobile networks," in *International Conference on Distributed Computing Systems (ICDCS 2007)*. IEEE Computer Society, 2007, p. 1.

[7] S. Dolev, S. Gilbert, L. Lahiani, N. A. Lynch, and T. Nolte, "Timed virtual stationary automata for mobile networks," in *Principles of Distributed Systems, 9th International Conference, OPODIS 2005*, ser. Lecture Notes in Computer Science, vol. 3974. Springer, 2006, pp. 130–145.

[8] R. Fan, I. Chakraborty, and N. Lynch, "Clock synchronization for wireless networks," in *OPODIS 2004: 8th International Conference on Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 3544. Springer, 2005, pp. 400–414.

[9] S. Umeno and N. Lynch, "Safety verification of an aircraft landing protocol: A refinement approach," in *Proc. of HSCC'07, Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, vol. 4416. Springer, 2007, pp. 557 – 572.

[10] R. Alur and D. L. Dill, "A theory of timed automata." *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[11] T. A. Henzinger, "The theory of hybrid automata," in *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 1996, p. 278.

[12] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997. [Online]. Available: citeseer.ist.psu.edu/larsen97uppaal.html

[13] T. Henzinger, J. Preussig, and H. Wong-Toi, "Some lessons from the HYTECH experience," in *Proc. of the 40th Annual Conference on Decision and Control*. IEEE Computer Society Press, 2001, pp. 2887–2892.

[14] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2." in *Proc. of CAV 2004*, ser. Lecture Notes in Computer Science, vol. 3114. Springer, 2004, pp. 496–500.

[15] B. Dutertre and M. Sorea, "Timed systems in SAL," SRI International, Tech. Rep. SRI-SDL-04-03, 2004.

[16] ——, "Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata." in *Proc. of FORMATS/FTRTFT 2004*, ser. Lecture Notes in Computer Science, vol. 3253. Springer, 2004, pp. 199–214.

[17] G. M. Brown and L. Pike, "Easy parameterized verification of biphase mark and 8N1 protocols." in *Proc. of TACAS 2006*, ser. Lecture Notes in Computer Science, vol. 3920. Springer, 2006, pp. 58–72.