In Proc. of the 6th ACM & IEEE Conference on Embedded Software (EmSoft'06).
Revised version.

An Analysis Framework for Network-Code Programs[*]

Madhukar Anand, Sebastian Fischmeister, and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
anandm@cis.upenn.edu, sfischme@seas.upenn.edu, lee@cis.upenn.edu

ABSTRACT

Distributed real-time systems require a predictable and verifiable mechanism to control the communication medium. Current real-time communication protocols are typically independent of the application and have intrinsic limitations that impede customizing or optimizing them for the application. Therefore, either the developer must adapt her application and work around these subtleties or she must limit the capabilities of the application being developed.

Network Code, in contrast, is a more expressive and flexible model that specifies real-time communication schedules as programs. By providing a programmable media access layer on the basis of TDMA, Network Code permits creating application-specific protocols that suit the particular needs of the application. However, this gain in flexibility also incurs additional costs such as increased communication and run-time overhead. Therefore, engineering an application with network code necessitates that these costs are analyzed, quantified, and weighted against the benefits.

In this work, we propose a framework to analyze network-code programs for commonly used metrics such as overhead, schedulability, and average waiting time. We introduce *Timed Tree Communication Schedules,* based on timed automata to model such programs and define metrics in the context of deterministic and probabilistic communication schedules. To demonstrate the utility of our framework, we study an inverted pendulum system and show that we can decrease the cumulative numeric error in the model's implementation through analyzing and improving the schedule based on the presented metrics.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Measurement techniques

General Terms: Measurement, Theory.

Keywords: real-time networking, scheduling, network code.

[*]This research is supported in part by NSF CNS-0410662, NSF CNS-0509327, NSF CNS-0509143, ARO W911NF-05-1-0182, and OEAW APART-11059.

1. INTRODUCTION

Distributed real-time systems consisting of several nodes connected via a shared communication medium require a predictable and verifiable mechanism to control the communication medium. By a predictable mechanism, we mean that communication between two parties happens in bounded time. By a verifiable mechanism, we mean that we can statically analyze, whether the specified requirements of the application will be met under the communication protocol being used. The goal of real-time communication protocols is to provide such mechanisms to applications. Examples of systems that need real-time communication protocols are industrial process control, integrated medical device networks, airplanes, and cars. Example communication protocols are diverse fieldbus systems, the Communication Area Network (CAN) [6], the Time-Triggered Protocol (TTP) [12], FlexRay [10], PowerLink Ethernet [5], and FTT-CAN [8].

Current real-time communication protocols are typically independent of the application. All the protocols mentioned above provide means for predictable communication. Some of them such as TTP or the static parts of FlexRay and FTT-CAN also permit static verification. However, although it is possible to realize a distributed real-time system with any of these protocols, not all of them always meet the needs of a particular application satisfactorily. These protocols have intrinsic limitations that impede customizing or optimizing for the application. Therefore, either the application developer has to adapt her application to work around these subtleties or she has to limit the capabilities of the application being developed.

To overcome these limitations, we have proposed network code [9], which permits creating application-specific protocols by providing a programmable media access layer. Network code is an executable communication abstraction to specify predictable and verifiable communication for distributed real-time applications. At the programming level, it provides control of timing, values, communication resources, and dynamic behavior.

Network code advocates creating application-specific media access control and stateful schedules based on application requirements. For example, if it is known that one machine on an assembly line is always turned off during lunch hours, then during this time its TDMA slot in the cycle can be assigned to another node. Or in a system with primary and multiple backups, if the primary has transmitted the data, then the secondaries do not need to send their data.

However, this gain in flexibility comes at a cost. Interpretation of network-code programs takes time depending on the size of the program [9]. Network-code programs also requires space in the ROM and RAM. This is especially crucial when programing micro-controllers with only a few bytes of program memory or RAM.

Such analysis has been done for traditional TDMA systems. For a particular protocol, such as FTT-CAN, the work [1] presents schedulability conditions for synchronous and asynchronous traffic. For more general TDMA communication, a necessary condition for round-robin TDMA and a sufficient condition for specific resource reclamation policies is specified in [11]. Also overhead has been considered in relation with schedulability [16]. Other metrics as overhead, average wait time, and average cycle length are also standard. However, network-code programs are more expressive than TDMA, so these metrics and analysis results are inapplicable and need to be defined in the context of network code.

Problem Definition. The network-code program represents a trade-off between different factors such as communication overhead or average cycle length. To make design decisions, such programs need to be analyzed and the benefits weighed against the incurred costs.

We provide a framework to analyze network-code programs for commonly used metrics such as schedulability, mean waiting time, thoughput, overhead, etc. We introduce *Timed Tree Communication Schedules*, based on timed automata to model network-code programs and present the metrics in the context of deterministic and probabilistic communication schedules. This includes for example, (1) a necessary condition for checking schedulability of a demand set given a network-code program based on CTL, (2) a sufficient condition for checking schedulability of a demand set given a network-code program in linear time w.r.t. the schedule length, and (3) the analysis of average waiting times based on queueing models.

We show the utility of the framework to model the network-code program by performing a case study with an inverted pendulum system. The inverted pendulum system has served as a benchmark problem in control systems and represents a large class of control problems. We model the inverted pendulum as a hybrid system and evaluate the effectiveness of different schedules using the proposed framework.

2. NETWORK CODE

Network code is an executable communication abstraction and allows application-specific protocols by providing a programmable media access layer on the basis of TDMA. Network code is basically a small program, specifying, when and what is going to be transmitted and received. For more details about the model and its assumptions see [9].

The instruction set consists of nine instructions, which fall into the categories of flow control, data control, and error handling. Especially interesting is the *if* instruction, which permits programming on-the-fly choices in the communication schedule. The condition in the *if* instruction is referred to as *guard*. Using guards, multiple communication instances may be scheduled for the same slot and the guard eventually grants the slot to at most one, on-the-fly.

In the current implementation, an interpreter called the network-code machine (NCM) executes the program. An NCM exists for RTLinuxPro 2.1 on top of the FSMLab's LNet driver for Ethernet and recently also for the PIC18F2X80 micro-controller on top of CAN.

Figure 1 shows a simplified overview of the system highlighting the important parts for this paper. A more elaborate description can be found in [9]. The application sits

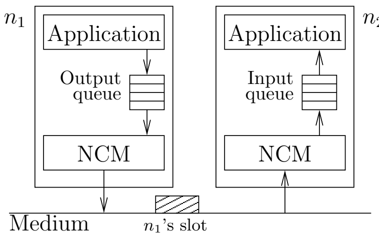


Figure 1: Queue overview for one application.

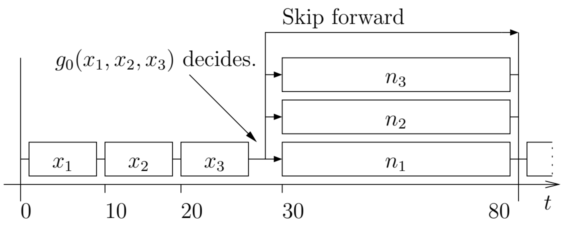


Figure 2: A schedule with one choice g_0.

on top. Each application has an application-specific message output queue. The application enqueues messages in this queue. The network-code program specifies, at what time messages are taken from the output queue and transmitted on the medium. The NCM interprets the program, and whenever the node has access to the medium, it will transmit as many messages as possible.

On the receiving side, the reverse is happening. The NCM receives data. The program specifies, into which application's input queue the NCM should enqueue the received data. Finally, the application can access the data from its input queue.

Example 1. *We want to realize a schedule in which three nodes vote based on three values x_1, x_2, and x_3. The highest value wins the voting, in a tie the value with the lower identifier i in x_i wins the voting. The node, which sends the winning value, is allowed to use the slot that follows the voting. If a node wants to pass one round, then it will send the value* 0*. If all nodes pass that round, then a new round will start immediately. Figure 2 implements this and guard g_0 implements the voting.*

For this example, we assume a global synchronized clock and that node n_1 sends value x_1. Listing 1 implements Figure 2's schedule. The parameters, which are unimportant for this paper, are replaced with _. First, n_1 transmits the value x_1. Second, it receives x_2 and x_3. Then, it evaluates g_0, and if g_0 returns true, then it will jump to label L0 *and transmit further data. Otherwise, it will receive another node's transmission. For details about the instructions see [9].*

```
L0: xsend ( _, _, X1, _ )
    wait ( 20 )
    receive ( _, X2 )
    wait ( 10 )
    receive ( _, X3 )
    if ( g0, L1 )
    wait ( 50 )
    receive (_, OTHERS )
    goto ( L0 )
L1: xsend ( _, _, DATA, _ )
    wait ( 50 )
    goto ( L0 )
```

Listing 1: Program for n_1.

3. ANALYSIS FRAMEWORK

In the following section, we present the analysis framework. Under this framework, we model network-code programs as a *timed tree communication schedule,* which is based on timed automata with a set of constraints and attributes. We then use this formalism to describe the various metrics for network-code programs.

3.1 Timed Tree Communication Schedules

Tree schedules have been introduced in [9]. However, that definition considers only isochronous cycles (i.e., cycles of equal length in time) and treats slots dedicated to guard data and application data equally. Here, we extend this definition and permit anisochronous cycles (i.e., cycles with different length) and distinguish between application-specific and guard-specific slots. The anisochronous extension makes our model more general and allows us to analyze a larger group of schedules and distinction between guard and application data is needed for calculating different metrics as explained in the following sections. Our definition here models the tree schedule based on timed automata [2].

Before we define the tree schedules formally, we introduce broadcast communication. A broadcast communication b is a set of tuples $\langle n_x, n_i \rangle$ where n_x is the sender and n_i is the receiver. In a broadcast communication b, each tuple has the same sender.

DEFINITION 1. *A* timed tree communication schedule *(TTCS) Ω is a tuple $\langle V, v_0, V_F, l, sl, K, E, M \rangle$ where*

- *V is a set of locations,*
- *$v_0 \in V$ denotes the initial location,*
- *$V_F \subseteq V$ denotes the set of final locations,*
- *l is a mapping $l : V \rightarrow \{app, guard, \perp\}$ with $L(v)$ that maps location to labels, The set of labels represents whether a location is application-specific (app), guard-specific (guard), or unspecified ($\perp$).*
- *sl is a mapping $sl : V \rightarrow B$ that maps a location to a broadcast communication associated with that location,*
- *K is a set of clocks(clk) with $|K| \geq 1$,*
- *E is a set of tuples $\langle s, g_x, \lambda, s' \rangle$ representing transitions from location s to location s'. The guard g_x is an enabling condition and λ is a set of updates on clock values. Also, the set of transitions E is such that $s \neq s^{'}$ for every tuple and there are no back edges other than from $v_f \in V_F$ to v_0.*
- *M is an extensible set of abstract metric-specific mappings that can be defined to help evaluate the metric. For instance, one such mapping could map guards with probabilities, and another mapping could map locations with the computation overhead.*

Note that an important property in a tree schedule (TTCS) is that there are no self loops or back-edges other than the transition from the leaf of every branch to the initial node v_0. Figure 3 shows a tree schedule as a directed acyclic graph (DAG). The dashed edge marks the reset to the start of the next round. For applying our analysis framework, such a DAG has to converted into a tree by duplicating nodes as necessary.

EXAMPLE 2. *Consider the schedule in Example 1. To represent the schedule as a timed tree communication, we define the tuple $\langle V, v_0, V_F, l, s, K, E, M \rangle$ as in Definition 1:*

- $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$

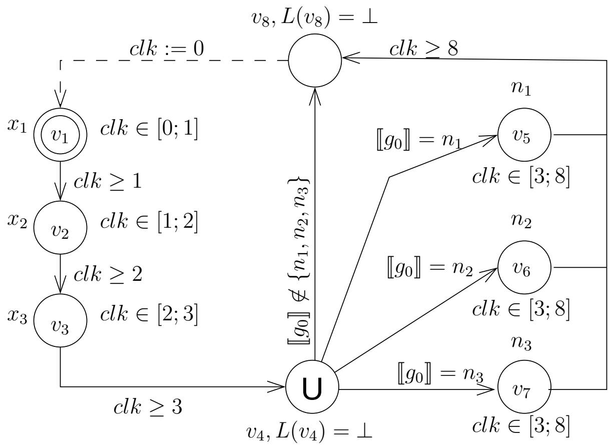


Figure 3: The schedule of Example 2 as a DAG. The dashed edge marks the reset to the start of the next round.

- *$v_0 = v_1$,*
- *$V_F = \{v_1\}$,*
- *l is $l(\alpha) = guard$ with $\alpha \in \{v_1, v_2, v_3\}$, $l(\alpha) = app$ with $\alpha \in \{v_5, v_6, v_7\}$, $l(\alpha) = \perp$ otherwise,*
- *sl: $sl(v_1) = sl(v_5) = \{\langle n_1, n_2 \rangle, \langle n_1, n_3 \rangle\}$, $sl(v_2) = sl(v_6) = \{\langle n_2, n_1 \rangle, \langle n_2, n_3 \rangle\}$, $sl(v_3) = sl(v_7) = \{\langle n_3, n_1 \rangle, \langle n_3, n_2 \rangle\}$, $sl(v_4) = sl(v_8) = \emptyset$ and a mapping of v_1 to x_1, v_2 to x_2, v_3 to x_3, v_5 to n_1, v_6 to n_2, and v_7 to n_3,*
- *$K = \{clk\}$,*
- *E is as shown in Figure 3, and*
- *$M = \emptyset$*

The guard g_0 is a function described as:

$$g_0(x_1, x_2, x_3) = \begin{cases} n_1 & x_1 \geq x_2 \wedge x_1 \geq x_3 \\ n_2 & x_2 > x_1 \wedge x_2 \geq x_3 \\ n_3 & x_3 > x_1 \wedge x_3 > x_2 \\ \perp & x_1 = x_2 = x_3 = 0 \end{cases}$$

In the example, first, n_1 communicates the value x_1, then n_2 communicates x_2, and n_3 communicates x_3. Second, each node evaluates guard $g_0(x_1, x_2, x_3)$. Depending on the result, one of the nodes will get the slot $[3, 8]$.

3.2 Conversion of Programs to TTCS

A network-code program is a sequence of network-code instructions, and to analyse it in our framework, we have to convert it to a TTCS. Network code by itself is a more abstract representation of the communication than what we have in the TTCS. Therefore, we need to annotate the program before we can convert it. The annotation is then put into the set of attributes for a node.

For the basic TTCS the metrics presented in this paper we need to annotate (1) the semantic meaning of communication instances, (2) model guards as switch conditions, and (3) evaluate the guard function's execution times. The unannotated program describes only when, what will be communicated. It is ignorant of the data's type and semantics in the application. We need to provide the value for the mapping $l(v)$ as annotation. Network code represents guards in C functions. We need to annotate the *if* statements with expressions, which model the guard and can be put into the automaton. If it is impossible to do so, we can use a probabilistic model as shown later. Finally, we need to annotate the execution time of the guard function.

The conversion of the annotated program is straightforward and similar to the conversion to VERSA presented in [9]. We interpret the program without performing actions on the shared medium. Each *send* instruction is encoded as a location. Each *if* statement is encoded with two successor locations. The annotation specifies the switch condition. *Future* instructions define how long the automaton delays in the current location. Afterwards, we can create a cross product of all the created automata and add invariants to each location, so the location must be left as soon as one leaving transition is enabled. For example, one node has one location for slot one and one for slot four. Another has a location for slot two and three. The automaton enters the location for slot one but leaves it as soon as possible for slot two. Otherwise, it would miss these slots.

3.3 Probabilistic Communication Schedules

In some applications, it may be expensive or even impossible to analyze the guards a priori. If transitions cannot be anticipated, we will be unable to compute reachable sets and therefore, provide only limited analysis. However, even with this restriction, it may be possible to attribute a transition probability to a switch. Once the transition probabilities are known, we analyze the network and support a different set of metrics such as average cycle length or average waiting time.

The transition probabilities on transitions can be obtained by calculating (see Equation (9) in the case study in Section 8) or by profiling the application at hand. Although with profiling, we only get absolute probabilities, we can normalize them (from a particular state) to generate the required probabilities.

To reap the benefits of measuring probabilities, we can replace the guards with functions, which decide on paths depending on the transition probabilities. This causes a switch to be taken purely on the result of a random experiment (such as tossing a fair coin). Consequently, we can provide analysis metrics, which require probabilistic analysis of the guards. Example analysis metrics are average cycle length and average waiting time.

We formally capture this idea as *probabilistic timed tree communication schedule* (pTTCS).

DEFINITION 2. *A* probabilistic timed tree communication schedule *(pTTCS)* Ω_p *is defined as a tuple* $\langle V, v_0, V_F, l, sl, K, E', M\rangle$ *where* V, v_0, V_F, l, sl, K *are as defined in Definition 1 and,* $E' = \langle s, a, \pi_x, \lambda, s'\rangle$ *where* $\pi_x = \pi(E')$ *is the probability of transition* E'*, and* $M = \{\pi\}$ *where* $\pi : E' \to [0, 1]$ *represents the probability of a transition.*

In the following sections, we define a set of analysis metrics, which can be used to analyze network-code programs.

4. OVERHEAD

Network code introduces two types of overhead: slot overhead and guard overhead. Slot overhead is the number of slots required to maintain consistent on-the-fly decisions throughout the network. Guard overhead is the amount of execution time required to perform on-the-fly decisions.

Overhead is an analysis metric because guard overhead adds to the system run-time overhead and slot overhead increases the cycle length. If there is very high guard overhead, or if the system overshoots its computation-time budget, overhead can give clues what need to be changed. Similarly, if any of the deadlines are not met, or there is a perceivable service latency, slot overhead should be reduced. To compute both the types of overhead, we first define a run of the TTCS.

DEFINITION 3. *A* run r *is a sequence of progression of states of a TTCS* $\langle v\rangle_n$ *of locations with* $v_0 \to v_1 \to \ldots \to v_n$*, where* $\to$ *represents a valid transition of the TTCS.*

The slot and guard overheads can now be defined for a particular run. The *slot overhead* of a run r is the total sojourn time of guard-specific locations in relation to the sojourn time in application-specific locations. Formally,

DEFINITION 4. *If* $G_x = \{v_i | v_i \in r, l(v_i) = x\}$ *is the set of locations with a label* x*, and* $sojourn(v)$ *is the amount of time TTCS spends in location* v*, then, the* slot overhead $o_s(r)$ *is defined as,*

$$o_s(r) = \frac{\sum_{v_i \in G_{guard}} sojourn(v_i)}{\sum_{v_i \in G_{app}} sojourn(v_i)} \quad (1)$$

The maximal/minimal guard overhead for a TTCS Ω *is given by* $\max_r o_s(r)/\min_r o_s(r)$*. The average guard overhead for* Ω *is given by* $\sum_r p(r) \cdot o_s(r)$*, where* $p(r)$ *is the probability of a run* r*.*

The *guard overhead* of a run r is the amount of computation time required along executing a path in relation to its duration.

DEFINITION 5. *If* $sojourn(v)$ *is the amount of time TTCS spends in location* v*, and* $WCET(g)$ *represents the worst-case computation time of guard* g*, the* guard overhead $o_g(r)$ *is defined as,*

$$o_g(r) = \frac{\sum_{g \in E} WCET(g)}{\sum_{v_i \in r} sojourn(v_i)} \quad (2)$$

where E *is the set of all transitions in the TTCS* Ω*. The maximal/minimal guard overhead for* Ω *is given by* $\max_r o_g(r)$ */*$\min_r o_g(r)$*. The average guard overhead for* Ω *is given by* $\sum_r p(r) \cdot o_g(r)$*, where* $p(r)$ *is the probability of a run* r*.*

5. SCHEDULABILITY

Given a specified node's communication demand and a tree schedule Ω, the schedulability condition checks whether this demand can be met, i.e., whether this particular node has enough slots to communicate in the specified time frame.

The schedulability condition allows the developer to check a schedule against high-level requirements. For example, if a specific node hosts a critical application such as a plant controller that must react within x time units, then the schedulability condition can be used to determine whether this demand can be met.

Before we describe the schedulability conditions for network-code programs, we introduce some notation. The demand of one node n_i is specified by the tuple $df(n_i) = \langle o, c, d, p\rangle_i$ with the offset o_i, the communication time c_i, the deadline d_i, and the period p_i. We assume that for each demand $df(n_i)$, $o_i + c_i \leq d_i \leq p_i$. We only consider demands, which are released at the beginning of each period. The set of demands for all nodes is represented by DF.

We are now ready to analyze schedulability for network-code programs. First, we describe an algorithm that checks for schedulability for a given node n given a TTCS Ω and a set of demands DF. It essentially implements a depth first traversal of the underlying graph of the TTCS Ω, and checks if the node n's demands are met.

Algorithm 1 *dfs(v,κ,s,n,df,Ω)* :Algorithm to check sufficiency of Ω given *df*.

```
Require: $\Omega = \langle V, v_0, V_F, l, sl, K, E, M\rangle$
Require: $df(n) = \langle o, c, d, p\rangle$
Require: $v \in V$
      if $v$ supplies demand $df(n)$ then
        $t \leftarrow 1$
        while $t * sl < sojourn(v)$ and $\kappa + t * sl < d$ do
          // Multiple transmissions per slot
 5:       if $\kappa + (t-1) * sl \geq o$ then
            $s \leftarrow s + 1$
          end if
          $t \leftarrow t + 1$
        end while
10:   end if
      // Update passed time.
      $\kappa \leftarrow \kappa + sojourn(v)$
      if $\kappa \geq d$ and $s < c$ then
        // Deadline passed, insufficient supply.
15:     return $false$
      end if
      if $s \geq c$ then
        // Deadline passed, sufficient supply.
        return $true$
20:   end if
      for all locations $v_x \in V$ adjacent to $v$ do
        // DFS recursion.
        if $dfs(v_x, \kappa, s, n, df, \Omega) = false$ then
          return $false$
25:     end if
      end for
      return $true$
```

The exact condition for Line 1 is $l(v) = a$ and $\langle c_i, c_j\rangle \in sl(v) : c_i = n$: the slot has to be dedicated to application-specific data and it has to supply the node n. κ is an integer clock value measuring the amount of time passed since the start until the demand's deadline. t is also an integer clock value measuring the time inside a slot. All other symbols have the meaning as defined before.

The above algorithm gives a sufficient condition for schedulability because it ignores guards and hence looses information within a cycle and across cycles. See Example 3 for an example. Nevertheless, since the algorithm runs in linear time, it may still be attractive in many applications to quickly check for schedulability.

THEOREM 1. *A demand set DF is schedulable given a tree schedule Ω, if $\forall n_i : dfs(v_0, 0, 0, n_i, df, \Omega) = true$, where dfs is implemented as in Algorithm 1 and $df(n_i) \in DF$.*

To provide a necessary condition for schedulability, we have to consider guards in the TTCS and evaluate whether the demands are met over all runs of the automata. We therefore encode the schedulability condition as a CTL expression and model-check the TTCS. A set of demands DF is schedulable given a TTCS Ω, *iff* the model-checker returns *true* for schedulability queries for all nodes. This idea is captured formally in the theorem below.

THEOREM 2. *A demand set DF scheduled using a tree schedule Ω is schedulable, iff $\forall \langle o, c, d, p\rangle_i = df(n_i) \in DF$: $AG\left(s_i \geq \lfloor\frac{\kappa}{p_i}\rfloor c_i + \delta_i\left(\kappa - \lfloor\frac{\kappa}{p_i}\rfloor p_i - o_i\right)\right)$ where,*

$$\delta_i = \begin{cases} 1 & \text{if } c_i \geq \kappa - \lfloor\frac{\kappa}{p_i}\rfloor p_i - o_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

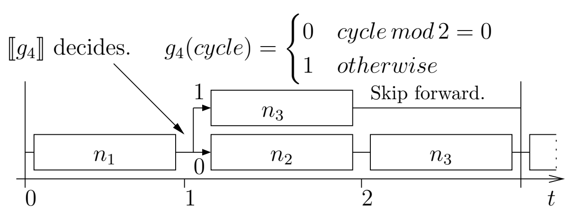


Figure 4: A tree schedule with one on-the-fly choice.

s_i is the number of application-specific slots for n_i in the interval $[0, \kappa]$, and other variables are as defined before.

We assume that the scheduler executes the TTCS as specified. By that it is assured that if the demand is less than the supply, then the schedule is feasible.

EXAMPLE 3. *Figure 4 shows a schedule with one on-the-fly choice at time $t = 1$. Consider the demand $df(n_1) = \langle 1, 3, 8, 8\rangle$. Applying Theorem 1, it will report that $df(n_1)$ will miss the deadline according to slot assignment $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_1$. However, applying Theorem 2 will result that it is schedulable. The reason is that guard g_4 will decide that n_1 gets two slots every other cycle. Thus, the path is $n_1 \rightarrow n_3 \rightarrow n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_1 \rightarrow n_3 \rightarrow n_1$ and the demand is satisfied.*

6. AVERAGE CYCLE LENGTH

Traditionally, in real-time communication protocols like TDMA, the cycle length is fixed. However, network code permits variable cycle lengths. In this section, we will define the metric for computing the average cycle length given a TTCS. The average cycle length measures how much time passes between repetitive behavior on the schedule, i.e., the average time it takes for the schedule to return to location v_0.

This metric is important in applications where certain actions happen once every cycle. For example, to estimate how often a clock update occurs that is always updated at the beginning of the cycle or to estimate regular actuator updates in control applications (see Section 8 for an elaborate example). We can also use this metric to estimate how often certain guard-enabled actions are performed.

DEFINITION 6. *Given a TTCS Ω with the mapping $\pi : E \rightarrow [0, 1], \pi \in M$ representing probabilities of transitions, the* average cycle length *can be estimated as,*

$$\sum_{v_i \in V_F} p_{v_i} \cdot dt(v_i, v_0) \qquad (3)$$

where v_0 represents the initial node, $dt(u, v)$ is the time it takes for the schedule to reach u from v, and $p_{v_i} = \pi(v_0, v_1) \ldots \pi(v_k, v_i)$ where $v_0, v_1, \ldots, v_k, v_i$ forms a path in the automaton from the initial node v_0 to the node v_i.

EXAMPLE 4. *Consider the schedule in Figure 4 of the Example 3 with the probability of guard transition to τ_3 with probability $\frac{3}{4}$ and τ_2 with probability $\frac{1}{4}$. Also consider that the n_3 in the upper path is only single slot in length. Then, the average cycle length would be $\frac{3}{4}(1 + 1) + \frac{1}{4}(2 + 1) = \frac{9}{4}$. This is obtained by summing up the metric over nodes v_3 and v_4 as these are the nodes that have a transition back to the start node v_1.*

The average cycle length can be computed by performing a depth-first search of the schedule and checking whether there is edge leading back to the initial node (v_0) and incrementing the probabilities.

7. AVERAGE WAIT TIME

The average wait time describes how long a specified node has to wait from the time it arrives to the time it actually is serviced (i.e., when it gets a communication slot in the network). This is an important metric both from the perspective of quality of service, and also to compute the buffer size on each node, so that messages are not dropped at the buffers in the system. The average wait time is especially important for probabilistic network-code programs where there is only a probability of a path being taken in the TTCS.

We expect that we are given an application that generates different events. These events in turn generate messages, which need to be sent on the network. We assume that there are two types of events: independent and dependent ones. Independent events arrive given a distribution. Dependent events occur given the conditional probability and the probability the event on that they depend on. We assume that arrival and conditional probabilities are known. For instance, such a model can be presented in the form of a dependency graph. Such a dependency graph can be generated from the knowledge of dependencies and probabilities of different events. In addition to the message arrival at nodes in the network, we assume a probabilistic TTCS Ω_p, which specifies the application's schedule. We also assume that there is a known mapping from messages to slots in the communication schedule.

Terminology. When messages arrive, we assume that they will be enqueued in a queue. The message at the head of the queue will be serviced in the slot assigned to the node holding the message. We refer the time spent in the queue until the message reaches the head of the queue as the *waiting time* (T_W). The *effective service time* (T_{ES}) is the total time spent at the head of the queue waiting for the slot and the time in the slot (T_S). The *effective waiting time* (T_{EW}) can be defined as $T_W + T_{ES} - T_S$.

In our analysis, we consider two types of systems: (a) event-triggered systems, in which messages are generated as a result of events, and (b) time-triggered system, in which the message arrival occurs at fixed points in time. For the analysis, the system is assumed to consist of a network of queues and we consider the service and waiting times once the system has reached a steady state. I.e., the waiting time may be different for the first few messages, but not so in a stabilized system. For event-triggered systems, independent events are considered to arrive as a Poisson process, i.e., the number of arrivals $N(t)$ of an event E in a finite interval t is given by $P\{N(E,t) = m\} = \frac{(\lambda t)^m}{m!}e^{-\lambda t}$. Further, to simplify the analysis, we consider infinite buffers. Although analysis with finite buffers would lead to a more precise result, it has been argued (for example, in [14]) that this error in analysis is marginal.

Waiting time analysis under event-triggered systems. Let us first consider the distribution of effective service times. Suppose an event E gets serviced in slots at locations $L = \{e_1, \ldots, e_m\}$ in the TTCS Ω_p. Consider one such location, say e_i. The effective service times experienced starting from this location depends on the next slot available for E starting from e_i. Now, it is possible that one of the descendants of e_i in Ω_p, say e_k will be one servicing E_i. If this is the case, then the effective service time will be $dt(e_i, e_k)$ where dt is as defined in Definition 6. The probability of reaching the slot e_j from e_i is $p_r(e_i, e_k) = \pi(e_i, v_1)\ldots\pi(v_r, e_k)$. We denote the set of all such slots to be $L_i \subset L$. If there is no such slot, then the schedule Ω_p will consist of a return to the initial location from one descendant locations and then, the service slot would be reached in the next cycle at any of the positions in $L \setminus L_i$. To calculate the effective service times for event E starting from the initial slot v_0, consider a location $e_k \in L$. The probability of reaching such a slot is $p_r(v_0, e_k) = \pi(v_0, v_1)\ldots\pi(v_r, e_k)$. The service time here is then $dt(v_0, e_k)$. The probability that no service slot would be reached is $(1 - \sum_{e_i \in L} p_r(v_0, e_i))$. If the average cycle length for schedules without any slot for E be $\bar{l}$, then the average effective service time starting from v_0, $\mu_E = \sum_{n\geq 0}\sum_{e_k \in L}(1-q_r)^n\, p_r(v_0,e_k)\cdot(n\bar{l} + dt(v_0,e_k))$ where $q_r = \sum_{e_j\in L} p_r(v_0, e_j)$. Therefore, the expected delay along this path would be $dt(e_i, e_j) + \mu_E$. If we denote the set of all descendant slots of e_i that are final locations before making a transition back to v_0 as F_i, then the average effective service time experienced at e_i is $\sum_{e_j \in L_i} p_r(e_i, e_j)\cdot dt(e_i,e_j) + \sum_{e_j\in F_i} p_r(e_i,e_j)\cdot(\mu_E + dt(e_i,e_j))$. We summarize the above discussion in the result below.

THEOREM 3. *Given a probabilistic TTCS Ω_p, if an event E_i is serviced at schedule locations $L^i = \{e_1^i, \ldots, e_m^i\}$ as specified in Ω_p, then the n^{th} moment s_n of the effective service times is given by,*

$$T_1 = \sum_{e_k^i \in L_k^i} p_r(e_j^i, e_k^i)\cdot dt(e_j^i, e_k^i)^n \quad (4)$$

$$T_2 = \sum_{e_j^i \in F_j^i} p_r(e_j^i, e_k^i)\cdot\left(\mu_{E_i} + dt(e_j^i, e_k^i)\right)^n \quad (5)$$

$$s_n(E_i) = \sum_{E_i}\sum_{e_j\in L^i} p(e_j)\cdot(T_1 + T_2) \quad (6)$$

$\mu_{E_i} = \sum_{h\geq 0}\sum_{e_k^i\in L^i}\left(1-q_r^i\right)^h\cdot p_r(v_0,e_k^i)\cdot(h\bar{l} + dt(v_0,e_k^i))$,
$q_r^i = \sum_{e_j^i \in L^i} p_r(v_0, e_j^i)$,

$$p_r(e_i,e_j) = \begin{cases} \pi(e_i,v_1)\ldots\pi(v_r,e_j) & Des(e_i,e_j) \\ \pi(e_i^j,v_1)\ldots\pi(v_f,v_0)\ldots\pi(v_r,e_j) & Otherwise \end{cases}$$

$p(e_j) = \frac{p_r(v_0,e_j)}{\sum_{e_i} p_r(v_0,e_i)}$,

where $Des(v_1, v_2)$ will be true, *if v_2 is a direct descendant of v_1 in Ω_p and* false *otherwise, and $\bar{l}$ is the schedule's average cycle length on other paths. Therefore, the mean ($\mu_X(E_i)$) and variance $\sigma^2(E_i)$ of effective service times (T_{ES}) are given by $s_1(E_i)$ and $s_2(E_i) - (s_1(E_i))^2$, respectively.*

Returning to the analysis of waiting times, the arrival of independent events is Poisson (and hence Markovian), the service times are general distribution function, we can model them as the queueing model $M/G/1$ for the independent events. Therefore, the variance in arrival times is $\sigma_T^2(E_i) = \frac{1}{\lambda_i^2}$. Now consider dependent events. Let us say that the dependent event E_j depends on an independent event E_i. Since the arrival process of E_i is Poisson, its inter-arrival times are exponentially distributed. The inter-arrival times (T_I) of dependent event E_j is given by the function, $P(T_I = t) = p_{ij}\cdot\lambda_i e^{-\lambda_i t}$. We can consider this as a $G/D/1$ queueing scheme. For a $G/G/1$, we have the following inequality on the waiting time, $W_q \leq \lambda\frac{\sigma_X^2 + \sigma_T^2}{2(1-\lambda\mu_X)}$ [7], where λ is the average rate of arrival, μ_X, σ_X are the mean and variance of the service times, and σ_T is the variance of the arrival time. The variance in arrival times is $\sigma_T^2(E_j) = \frac{p_{ij}}{\lambda_i^2}$ for a dependent event, and the rate of arrival is $\frac{\lambda_i}{p_{ij}}$.

Definition 7. *Given an event-triggered system comprising of events $\mathcal{E}$ such that the independent events arrive as a Poisson process, and these events are serviced by a probabilistic TTCS Ω_p, the* average waiting time *(T_W) of an event E_i satisfies,*

$$T_W(E_i) \leq \frac{\lambda_i^2 s_2(E_i)^2 + 1}{2\lambda_i(1 - \lambda_i s_1(E_i))} \tag{7}$$

where E_i is an independent event with arrival rate of λ_i, and

$$T_W(E_j) \leq \frac{p_j \lambda_i^2 s_2(E_i)^2 + 1}{2p_j\lambda_i(1 - p_j\lambda_i s_1(E_i))} \tag{8}$$

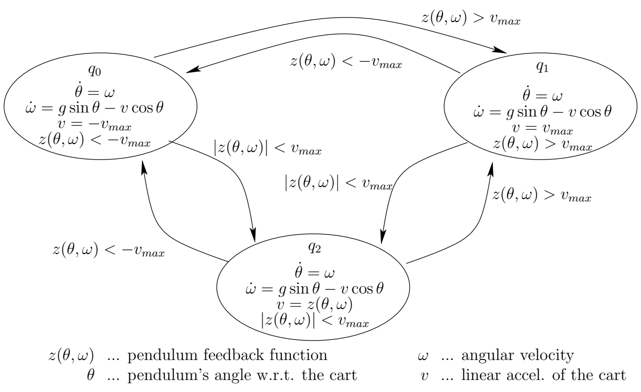
where E_j is a dependent event with interarrival times distributed as $p_j \cdot \lambda_k e^{-\lambda_k t}$ for some k. $s_1(E)$ and $s_2(E)$ are the average and variance of the effective service times as given in Theorem 3. The average waiting time for the whole system can be defined as, $\overline{T_W}(\mathcal{E}) = \frac{1}{|\mathcal{E}|}\sum_{E_i \in \mathcal{E}} T_W^{'}(E_i)$ where $T_W{'}$ represents the upper bounds of effective waiting times.

Example 5. *Consider the schedule in Figure 4 of the Example 3 with the probability of guard transition to τ_3 with probability $\frac{3}{4}$ and τ_2 with probability $\frac{1}{4}$. Again, assume that the slot for n_3 in the upper path is single slot in duration. Let us assume that the arrivals at n_1 is Poisson with rate λ_1. Then, the variance in arrival time for n_1 is $\frac{1}{\lambda_1^2}$. The mean service time for n_1 is $\frac{9}{4}$, and the variance is $(\frac{3}{4})2^2 + (\frac{1}{4})3^2 - (\frac{9}{4})^2 = \frac{3}{16}$. Let us analyze the waiting time for a dependent event n_3. Its mean arrival rate is $\frac{1}{\lambda}$, the mean service time would have to be calculated for positions of n_3 (on the path above and the path below). In both these cases, this is given by $(\frac{3}{4})1 + (\frac{1}{4})2$. Therefore the mean time is $\frac{5}{4}$. Its variance is then $(\frac{3}{4})1 + (\frac{1}{4})4 - (\frac{5}{4})^2 = \frac{3}{16}$. For n_2, the mean arrival time is $\frac{1}{4\lambda_i}$ the mean service time is $\sum_{n \geq 0}(\frac{3}{4})^n \frac{1}{4}(2 + 2 \cdot n) = 8$. The average waiting time for the entire schedule can now be calculated.*

Network Code under time-triggered systems. In a time-triggered system, the messages arrive at fixed points in time. Therefore, in this case, each node on the network will get a slot depending on the probabilities and the frequency of occurrence in the schedule. We can still consider the generation of a time-triggered message as an arrival which has a general distribution and service times also having a general distribution. Therefore, given the schedule Ω_p in the time-triggered case, we have a $G/G/1$ queueing system and the waiting time for the system can be bounded by $T_W(E) \leq \lambda\frac{\sigma_X^2 + \sigma_T^2}{2(1-\lambda\mu_X)}$ where the symbols have the usual meaning.

8. CASE STUDY

In our case study, we implement a control system for an inverted pendulum, which tolerates two independent value failures using a voting system. An inverted pendulum is essentially a pole mounted on a cart. The pole is free to rotate round on an axis, and the cart can move horizontally. The objective is to maintain the inverted pendulum in the upright position. The control strategy is modeled as a hybrid automaton that is described in Figure 5. The control model



q_0
$\dot{\theta} = \omega$
$\dot{\omega} = g\sin\theta - v\cos\theta$
$v = -v_{max}$
$z(\theta,\omega) < -v_{max}$
q_1
$\dot{\theta} = \omega$
$\dot{\omega} = g\sin\theta - v\cos\theta$
$v = v_{max}$
$z(\theta,\omega) > v_{max}$
q_2
$\dot{\theta} = \omega$
$\dot{\omega} = g\sin\theta - v\cos\theta$
$v = z(\theta,\omega)$
$|z(\theta,\omega)| < v_{max}$
$z(\theta,\omega) > v_{max}$
$z(\theta,\omega) < -v_{max}$
$|z(\theta,\omega)| < v_{max}$
$|z(\theta,\omega)| < v_{max}$
$z(\theta,\omega) > v_{max}$
$z(\theta,\omega) < -v_{max}$
$z(\theta,\omega)$ ... pendulum feedback function
θ ... pendulum's angle w.r.t. the cart
ω ... angular velocity
v ... linear accel. of the cart

Figure 5: Controller model of the inverted pendulum.

bases on previous works [15, 4]. The control parameter is the linear acceleration of the pivot v. This parameter bases on the pendulum's angle θ to the car and its angular speed ω. Initially, the automaton is in state q_0, which signifies acceleration to the extreme left. The state q_1 is the opposite of q_0 and q_2 describes the control in the stabilized position.

We want the system to tolerate two value failures of the variables θ and ω. We use standard five-way redundancy to tolerate such failures. In our particular case, we use five independent units reading θ and ω values to vote over possibly two value failures. Each unit broadcasts their result using a communication schedule. The control unit receives these results and performs a voting. To tolerate two failures in five results, a majority of at least three votes for one value will decide the voting and the control unit will adjust the acceleration v accordingly. We assume a failure rate of ten percent, i.e., in ten readings, one will be incorrect. We also assume failures to be transient.

8.1 The Setup

We implemented the hybrid systems control using Charon. Charon permits modular specification of interacting hybrid systems and supports automatic code generation [3]. Once the model is specified in Charon, the code generator for an agent takes a sampling step size as an input and produces code that approximates the continuous behavior of the model. We then simulate the code and monitor the runtime behavior.

We model communication between the controller and the replicated units via the sampling step size in our model. For example, in the standard TDMA system, each unit must report its value. The control unit always waits a full cycle, so the length of the TDMA cycle is the sampling step size. In the implementation, we ignore overhead introduced by clock synchroniation or computation time for reading values, adjusting values, and the voting. This can be incorporated as additional overhead to the cycle duration.

The expected behavior is that, since any step size introduces an error in the model, depending on the step size, the pendulum will collapse earlier or later due to the cumulative error. So, the step size is our control variable in the simulation, which is adjusted depending on the scenario we are running. Note, that the slot length and consequently the step size are chosen arbitrarily to show the point of the different schedules and shorten the time for the calculations.

For the case study, we consider a slot length of 0.0056 time units. Furthermore, we want unit u_1 to report a new value

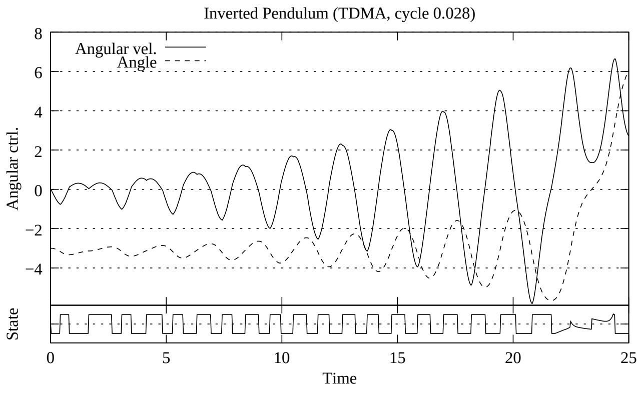


Figure 6: TDMA tolerating two value failures

every 0.020 time units and have the average cycle length below that number as well. We could have chosen other values, however, these show the point more clearly than others.

8.2 Step I - Traditional TDMA

The communication schedule using TDMA is that each unit transmits its read value. After each reported the value, the controller votes and updates the control input. Thus, the TDMA cycle is 0.028, i.e., five times the slot length of 0.0056.

Using a standard TDMA schedule, the pendulum collapses after 23.379 time units. Figure 6 shows the simulation results for standard TDMA. The top part of the diagram shows the angular values, and the bottom part shows the control input. In the top part, the dashed line shows the angular value θ. The solid line shows the angular velocity ω. In the bottom part, the dotted lines show the control input v.

The standard TDMA schedule can be modeled as a TTCS without an on-the-fly decision. So, we can apply our framework. Furthermore, we can check for scheduling by asking whether unit u_1 will be able to communicate one slot at least every 0.020 time units. Algorithm 1 shows it is not schedulable. This is confirmed by the average cycle length (see Equation (3)), which results to 0.028. Note, we chose u_1, but similar results go for u_2 and u_3. However, not for u_4 and u_5, because they have no guaranteed execution within one cycle.

8.3 Step II - Reducing Average Cycle Length

To meet the required 0.020 time constraint for unit u_1, we now will try to shrink the average cycle length. For this, we will increase the number of u_1 transmissions without compromising the application. First, we distinguish between necessary and unnecessary information. The first three slots (including u_1) are necessary for the application. The fourth slot only contains useful information, if the first three slots disagree on the value. The fifth slot only contains information, if from the previous four slots less than three agree on a value.

Using this insight, we now create a network-code program to minimize the number of unnecessary slots and thereby will (1) increase the number of slots for u_1 and (2) decrease the average cycle length. So, we implement a schedule, in which if after three or four reported measurements, the voting is already decisive (i.e., the number of votes of one particular value is greater than two), then the control unit will immediately apply the new value and all units will immediately start with a new cycle. Figure 7 shows the communication schedule for this case. We implement this in CHARON by simulating the network-code program, which implement this schedule, and by adapting the sampling step size accordingly to the current cycle length.

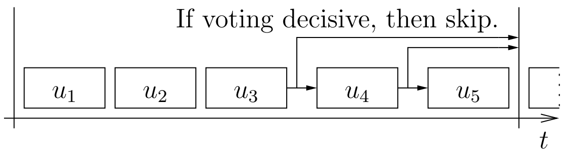


Figure 7: Tree schedule tolerating two value failures.

Listings 2 and 3 show the network-code programs without clock synchronization for these communication schedules. Listing 2 shows the controller unit's program. First, it waits until three units have sent their measurements. Second, in Line 5, a guard determines, whether the voting is already decisive. If it is decisive, then a new cycle will immediately start. If the voting is not decisive, then more data will be collected. Listing 3 shows the program for unit u_5. Its behavior is similar to the control unit's behavior, however, in Line 9, u_5 sends the data.

```
    L0: wait ( 3t )
        receive ( _, U1_DATA )
3       receive ( _, U2_DATA )
        receive ( _, U3_DATA )
        if ( votingDecisive, L0 )
        wait ( 1 )
        receive ( _, U4_DATA )
8       if ( votingDecisive, L0 )
        wait ( 1 )
        receive ( _, U5_DATA )
        goto ( L0 )
```

Listing 2: Program for the controller unit.

```
    L0: wait ( 3t )
        receive ( _, U1_DATA )
        receive ( _, U2_DATA )
4       receive ( _, U3_DATA )
        if ( votingDecisive, L0 )
        wait ( 1 )
        receive ( _, U4_DATA )
        if ( votingDecisive, L0 )
9       xsend ( _, _, MY_DATA, _ )
        future ( 1, L0 )
        halt ( )
```

Listing 3: Program for the unit u_5.

Using this network-code program, we can stabilize the pendulum for a much longer period than with the previous schedule (45 time units compared to 23). Figure 8 shows the simulation results. The top part shows the angular values. The solid line shows the angular velocity ω and the dashed line shows the angle θ. After a short time period of about 2 time units, the pendulum become stable, as the angular velocity is practically zero and the angle is π. The middle part shows the control input v. Once the pendulum becomes stable, it rapidly switches between acceleration modes (i.e., the black box in the figure) to keep it upright. The bottom part shows the number of slots per cycle. That means, if one error occurred in three slots, then the cycle will have four slots. If two errors occurred in four slots, then the cycle will have five slots. So for example at time 10, the cycle uses five

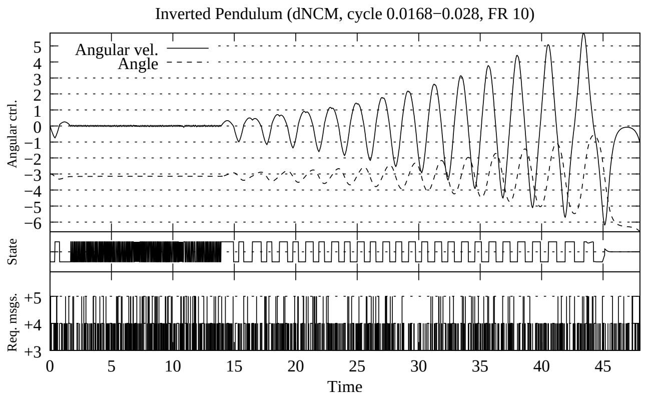


Figure 8: Program with deterministic guards.

slots, because two units reported a different value.

This new schedule resulted in a lower average cycle length and meets the demand for u_1. In the TDMA schedule, the average cycle length is 0.028. Using this schedule, the average cycle length is the average number of slots is 3.346 and the average cycle time is 0.0187376. Furthermore, Algorithm 1 returns that the demand of one slot per 0.020 time units is met (e.g., checking this for u_4 or u_5 results in a not schedulable).

However, evaluating the guard values introduces run-time overhead. In the next step, we try to reduce this overhead.

8.4 Step III - Reducing Guard Overhead

We want to reduce the run-time overhead by using probabilistic guards instead of the deterministic guards used before. To use the probabilistic model, we need to assign probabilities to transitions. When a system tolerates n failures, then the probability $P(x)$ of requiring $\gamma + x$ slots the schedule is:

$$P(x) = \binom{\gamma}{x} p^x (1-p)^{(\gamma-x)} \tag{9}$$

with $\gamma = \lceil \frac{2n+1}{2} \rceil$ defining the minimal number of slots, p defining the probability of any node failing, and $1 \leq x \leq \lfloor \frac{2n+1}{2} \rfloor$ defining the number of additionally required slots.

If $x = 1$, then we need one additional slot in our communication schedule, so one of γ slots contains a different value than the others. It contains a different value, when a failure occurred. For $\gamma = 3$ this is: $(1-p)(1-p)p + (1-p)p(1-p) + p(1-p)(1-p) = 3p(1-p)^2$. For an arbitrary γ, this is: $\gamma \cdot p \cdot (1-p)^{(\gamma-1)}$. Now for an arbitrary x, the probability is the sum of $p^x(1-p)^{(\gamma-x)}$ for all possible subsets of length x. Equation (9) follows.

We replace the deterministic guards with probabilistic guards as depicted in Figure 9 where γ is three. Equation (9) provides the values for the guards (i.e., $P(1) = 0.7323$ and $P(2) = 0.9726$). After three units have sent their value, a guard will decide based on the probability, whether the fourth unit's value will also be considered. Similarly, we treat unit u_5. If, at the end of the cycle, the voting is not decisive, then the control unit will omit the actuator updates for this cycle. We implemented this in CHARON by simulating the network-code program, which simulates this schedule and set step size to be the sum of all indecisive cycles plus the last decisive one.

The network-code program with probabilistic guards is similar to the one for the deterministic implementation. The only changes are that we replace the guard function *votingDecisive* in Line 5 with the guard function *pBelow7323.* This guard function returns *true,* if a selected random value is below 0.7323. We also replace the guard function in Line 8 with *pBelow9726.* This guard function returns *true,* if the selected random value is below 0.9726. We assume that all units draw the same random numbers (e.g., they have the same pseudo number generator and start with the same seed value). Otherwise, different nodes may choose different paths in the TTCS.

Statistically, this schedule does not have information overhead, because in the long run, the number of used packets matches the number of required packets. This schedule also reduces the run-time overhead, because the computation is now linear with complexity of the guard as drawing the random numbers is constant in time.

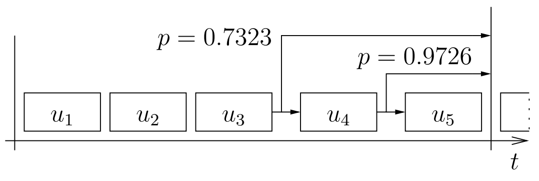


Figure 9: Tree schedule with probabilistic guards.

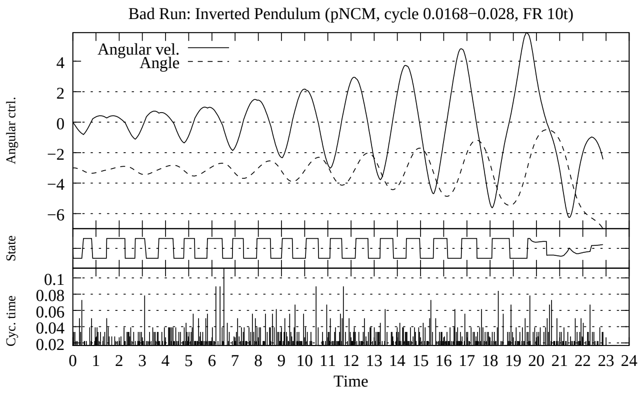


Figure 10: Program with probabilistic guards (a bad run).

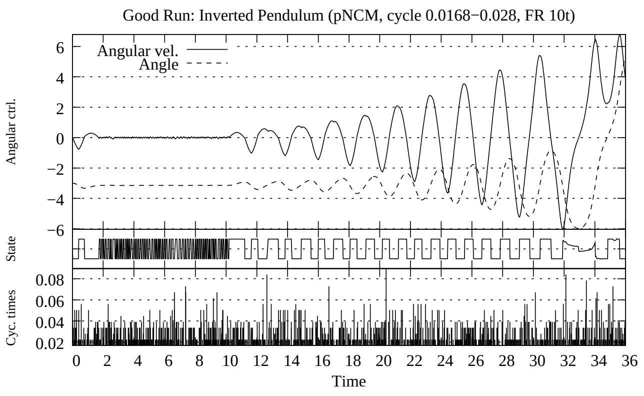


Figure 11: Program with probabilistic guards (a good run).

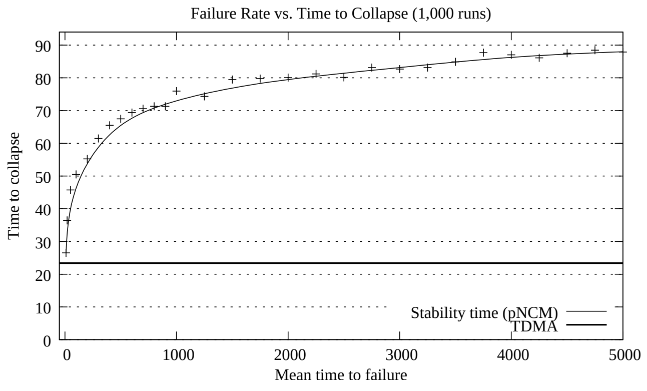


Figure 12: Pendulum stability with probabilistic guards with varying mean time to failure.

However, the simulation results are differential. There are very good runs and very bad runs. Figure 11 shows a good run, in which the pendulum remains stable for 34 time units. Figure 10 shows a bad run; the pendulum is stable for only about 23 time units. The top part shows the angular values. The solid line shows the angular velocity ω and the dashed line shows the angle θ. The middle part shows the control input v. The bottom part shows the time between two decisive cycles.

Given the large variation, we now determine, how the performance of the network-code program with probabilistic guards is affected by the failure rate. Therefore, we simulated a thousand runs with varying failure rate. Figure 12 shows the result. As the figure shows, with an increasing failure rate, the time to collapse using the program with probabilistic guards logarithmically approaches the upper limit.

Even at a failure rate of 10 percent, the program with probabilistic guards performs slightly better than the standard TDMA schedule shown before. The higher the rate, the better the program with probabilistic guard gets in comparison to TDMA, because TDMA assumes the worst case in each cycle and therefore stays at 23 time units. Furthermore, the failure rate of 10 percent is extremely high. Considering CAN messages with 20ms per message, we have a mean time to failure of 200ms. This is about *72 billion times* higher than the traditional numbers of 1,000 hours for transient node failures [13]. We chose such a low mean time to failure to show the point more effectively. The results are the same for high numbers.

9. CONCLUSIONS

Network code provide a flexible and expressive model of specifying and executing real-time communication schedules. In this paper, we have formalized network-code schedules as timed tree communication schedules (TTCS) based on timed automata, and have developed a framework with several common metrics for analyzing deterministic and probabilistic network-code programs. We have shown the framework's utility with an inverted pendulum case study. From the results of the study, we have found that the framework is applicable to model network-code programs and that the analysis helps improving schedules based on high-level requirements (e.g., minimizing the delay for actuator updates). Specifically in the case study, the network-code program with probabilistic guards has offered significantly shorter cycle lengths with a probabilistic guarantee of accurate values than the standard TDMA schedule. Consequently, the finally developed schedule has stabilized the pendulum much longer than the TDMA model even at very high failure rates.

While our framework is indispensable in weighing the costs versus the benefits under different metrics for an application developer, the framework also serves as an important step towards automatic generation of application-specific media access control based on higher-level requirements. In the future, we will explore the problem of optimizing TTCS considering sets of metrics.

10. REFERENCES

[1] L Almeida, P. Pedreiras, and J.A.G. Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Trans. on Industrial Electronics (TIE)*, 49(6):1189–1201, December 2002.

[2] R. Alur and D.L.Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183 – 235, 1994.

[3] R. Alur, F. Ivančić, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchial hybrid models. In *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.

[4] K. J. Astrom and K. Furuta. Swinging up a pendulum by energy control. *Automatica*, 1999.

[5] BERNECKER + RAINER Industrie-Elektronik Ges.m.b.H. *Ethernet Powerlink: Data Transport Services*, 5 edition, September 2002. White-Paper.

[6] Bosch. *CAN Specification, Version 2.* Robert Bosch GmbH, September 1991.

[7] Sanjay K Bose. *Introduction to Queueing Systems.* Kluwer Academic/Plenum Publishers, 2001.

[8] J. Ferreira, P. Pedreiras, L. Almeida, and J.A. Fonseca. The FTT-CAN protocol for flexibility in safety-critical systems. *IEEE Micro*, 22(4):46–55, July-Aug. 2002.

[9] S. Fischmeister, O. Sokolsky, and I. Lee. Network-Code Machine: Programmable Real-Time Communication Schedules. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006.

[10] FlexRay Consortium. *FlexRay Communications System — Protocol Specification*, June 2004. Version 2.0.

[11] K.A. Kettler, J.P. Lehoczky, and J.K. Strosnider. Modeling bus scheduling policies for real-time systems. In *Proc. of the Real-Time Systems Symposium (RTSS'95)*, pages 242–253, 1995.

[12] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997.

[13] H. Kopetz. The fault hypothesis for the time-triggered architecture. In *Proc. of the IFIP World Computer Congress*, 2004.

[14] Biplab Sikdar Shivkumar Kalyanaraman. Network layer performance modeling and analysis. `http://www.ecse.rpi.edu/Homepages/shivkuma/teaching/fall2001/ccn2001-slides.pdf`.

[15] Claire J. Tomlin. Hybrid Systems: Modeling, Analysis, and Control, Course in Stanford University. `http://www.stanford.edu/class/aa278a/lecture1.pdf`.

[16] E. Wandeler and L. Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *Proc. of the Asia and South Pacific Desing Automation Conference (ASP-DAC'06)*, Yokohama, Japan, January 2006.