On the Generation of Stateful Communication Schedules

Mahesh Balasubramaniam and Madhukar Anand
Department of Computer and Information Science
University of Pennsylvania
bmahesh@seas.upenn.edu,anandm@cis.upenn.edu

Abstract

Stateful Communication Schedules have been proposed with a view to provide flexible and predictable communication in real-time networks. Previous work in this area has largely focused on the verifiability of schedules and providing metrics such as schedulability, waiting time and overhead for the stateful schedules. In this report, we look at the problem of generation and representation of stateful communication schedules from an input requirement specification. Starting from configurations of possible messages and their release frequencies, we analyze schedulability under the Earliest Deadline (ED) and Deadline Monotonic (DM) scheduling strategies. If the messages are schedulable, then a stateful communication schedule for all the messages is generated. However, the generated schedule often results in a large labeled graph with overlapping nodes, providing an opportunity for compaction. We therefore discuss a few compaction strategies, touching briefly on the tradeoffs involved, and present an efficient algorithm for generating the most compact schedule. Finally, we conclude by describing the tool chain we have developed that produces a visual representation of the compact stateful schedules.

1 Introduction

Current real-time communication protocols such as the Communication Area Network (CAN) [4] and the Time-Triggered Protocol(TTP) [9] are typically independent of the application. Although these protocols provide a means for predictable communication, 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, stateful communication schedules implemented as network code [6, 3] have been proposed. The network code framework 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. A simple example of a stateful communication schedule is given in the Figure 1 below. In this schedule, x_1, x_2 and x_3 represent guard data and n_1, n_2 and n_3 represent the slots for nodes $1, 2$ and 3 respectively. The communication slot is alloted to either n_1, n_2 or n_3 based on the guard conditions. This strategy can be used to emulate the earliest deadline first scheduling by communicating the deadlines as guard data.

Prior work on these stateful communication schedules has largely focused on their verifiability and implementation via network code [6] or providing analysis metrics for them such as schedulability, service waiting time and computational overhead. In this report, we look at the problem of generation and representation of stateful communication schedules from an input requirement specification. The input specification in many real-time applications is often a simple model that describes the periodic release of messages along with their deadlines. More

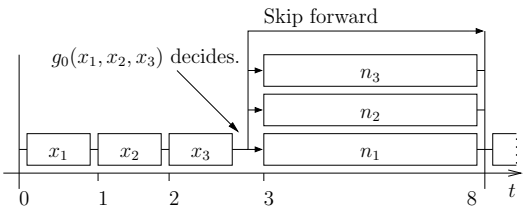

Figure 1. A schedule with one choice g_0.

complex models are starting to be popular in literature (c.f.,[10],[7]) but we will restrict ourselves to the periodic model here.

The focus of this project is to generate stateful communication schedules given a input requirement specification. To generate the schedule, we consider two scheduling strategies - Earliest Deadline (ED) and Deadline Monotonic (DM). These are essentially non-preemptive versions of EDF and RM scheduling respectively. Since the network communication is indivisible, we cannot preempt messages even if higher priority messages arrive while the message is being transmitted. If the messages are schedulable, then we generate the schedule for all the messages. This schedule is stateful because it executes a particular configuration of messages depending on the state of the network.

The schedule generated using either strategy often results in a large labeled graph with significant overlap between different configurations. It is therefore possible to compact the schedule so that the storage space is minimized. In general, a more compact representation can be achieved at the cost of more processing at runtime (resulting from including extra guard conditions). In this work, we discuss some tradeoffs involved and present an efficient algorithm for generating the most compact schedule.

In addition to generating the stateful communication schedule, we look at the problem of producing a representation of the schedule. A visual representation is more intuitive for the users and can be used to catch omissions and inconsistencies in the schedule. To this end, we have developed a tool chain that takes the input specification, checks schedulability, and generates a compact schedule that is presented to the user.

The rest of this report is organized as follows. In Section 2 we formally describe the input model, discuss the schedulability, schedule generation algorithms, and delve into compaction of schedules and the tradeoffs involved. In Section 3, we discuss the implementation of the tool chain and conclude with directions for future work.

2 Generation of Stateful Communication Schedules

Input Model We assume that the messages to be scheduled are given by the tuple $m = \langle m, p, pr, l \rangle$ where m represents the message identifier, p is the periodicity of the message, pr represents the priority and l the length of the message. Different messages that are scheduled together are specified as a configuration $C = \langle c, m_1, \ldots m_n \rangle$ where c is the configuration identifier and m_i are messages. Finally, the communication schedule is a set of configurations. Table 2 below shows an example input specification with different possible configurations.

Configuration ↓	Messages →		
c_1	$\langle m_1, 3, 1, 1 \rangle$	$\langle m_2, 3, 2, 1 \rangle$	$\langle m_3, 6, 1, 1 \rangle$
c_2	$\langle m_1, 6, 1, 1 \rangle$	$\langle m_2, 6, 2, 1 \rangle$	$\langle m_3, 3, 1, 1 \rangle$
c_3	$\langle m_3, 7, 1, 1 \rangle$	$\langle m_4, 2, 1, 1 \rangle$	$\langle m_5, 14, 2, 1 \rangle$

Table 1. Table showing a input specification with different possible configurations

The task here is to take the input model and use an appropriate scheduling algorithm to generate the schedule. We consider the non-preemptive versions of popular scheduling algorithms EDF and RM - the Earliest

Deadline and Deadline Monotonic scheduling. The algorithm for generating stateful schedule is given below as Pseudocode 1. We assume that there is a priority queue PrQ with methods $Push$ and Pop for pushing and popping from the queue. The method $AllocateSlots$ do the actual allocation and link the node with its parent, and $getLength$ method returns the length of a message. $isSchedulable$ returns true if the configuration is schedulable and false otherwise.

The algorithm is fairly straightforward. For every configuration, if it is schedulable, the algorithm initially enqueues all the messages. It starts allocating by popping the messages from the priority queue. The priority queue is implemented according to the scheduling algorithm used. For instance, with ED, a higher priority is assigned to a message with the earlier deadline. Similarly, with DM, a higher priority is assigned to the message with a shorter deadline. If there is no unique message, then the message with higher message priority is popped. After each message is schedule, the algorithm checks to see if any more messages are released in this duration and enqueues them also. The algorithm terminates at the LCM of the periods of all messages.

scheduleGen()

1: $AllocateSlots(root, \perp)$
2: $currNode \leftarrow root$
3: **for** Every $c_i \in C$ **do**
4: **if** $isSchedulable(c_i, RM/ED)$ **then**
5: $schLength \leftarrow 0$
6: **for** Every $m_i \in c_i$ **do**
7: $Push(PrQ, m_i)$
8: **end for**
9: **while** $schLength < LCM(p_j), \langle m, p, pr, l\rangle_j \in c_i$ **do**
10: $m \leftarrow Pop(PrQ)$
11: $currNode \leftarrow m$
12: $AllocateSlots(m, currNode)$
13: **for** Every $\langle m, p, pr, l\rangle_j \in c_i$ **do**
14: **if** $\exists t, t \in (schLength, schLength + getLength(m)], t \equiv 0 \pmod{p_j}$ **then**
15: $Push(PrQ, m_j)$
16: **end if**
17: **end for**
18: $schLength \leftarrow schLength + getLength(m)$
19: **end while**
20: **end if**
21: **end for**

Pseudocode 1: Stateful Schedule Generation

2.1 Schedulability

The main criteria for generating the schedule above is that the messages in a particular configuration be schedule. We revisit the schedulability criteria for ED and DM scheduling.

2.1.1 Earliest Deadline Scheduling

Schedulability conditions for non-preemptive ED are described by Zheng and Shin [11]. If all the messages are released at $t = 0$, they will be schedulable if the following conditions are met.

1. $\sum_{j=1}^{n} \frac{C_j}{T_j} \leq 1$
2. $\forall t \in S, \sum_{i=1}^{n} \lceil (t - d_i)/T_i \rceil^{+} C_i + C_p \leq t$, where $S = \cup_{i=1}^{n} S_i, S_i = \{d_i + nT_i : n = 0, 1, \ldots, \lfloor (t_{max} - d_i)/T_i \rfloor\}$ and $t_{max} = \max\{d_1, \ldots, d_n, (C_p + \sum_{i=1}^{n}(1 - d_i/T_i)C_i)/(1 - \sum_{i=1}^{n} C_i/T_i)\}$ where T_i, C_i, d_i are

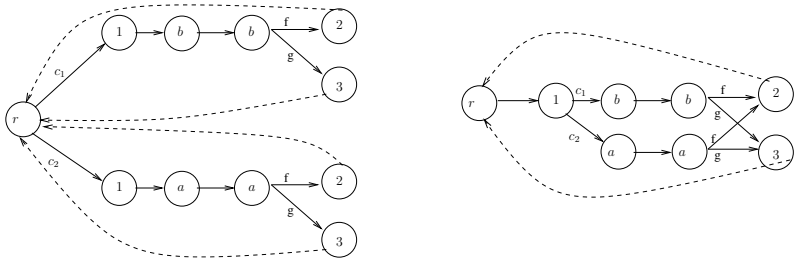

Figure 2. Compaction of generated schedule

the period, length, and deadline of message i, C_p is the length of the longest possible packet, and $\lceil x \rceil^+ = n$ if $n - 1 < x < n, n \in \mathbb{Z}^+$ and 0 otherwise.

The first condition ensures that the maximum utilization does not exceed capacity of the channel (assumed to be normalized to 1), and the second condition ensures that each message with deadline $\leq t$ can finish by t. For a more general condition involving phase offsets for messages, we refer the reader to Zuberi and Shin [12].

2.1.2 Deadline Monotonic Scheduling

For the non-preemptive case, a message i is feasible if all higher priority messages are feasible and i finds an opportunity to start transmission sometime during $[0, d_i - C_i]$. Therefore, to check for schedulability, we consider messages with a priority higher than that of message i and assume that they are the only messages in the system. If the network becomes idle in the interval $[0, d_i - C_i]$, then message i will be schedulable. If the messages are numbered according to their priority with $j = 0$ being the highest priority message, then i is schedulable if [8]:

- $\exists t \in S, \sum_{j=1}^{i-1} \lceil (t - \phi_j)/T_j \rceil C_j + C_p \leq t$ where $S = \{$set of all release times of messages $0, 1, \ldots, i-1$ through time $d_i - C_i\} \cup \{d_i - C_i\}$, and ϕ_j are the relative phase offsets.

We repeat this check on all messages in the configuration to check for schedulability.

2.1.3 Largest Schedulable Subset

When we are generating the stateful schedule, it is possible that not every configuration is schedulable. In that case, we either have the option of generating the schedule for a schedulable subset of messages or not consider that configuration at all. If we decide on picking a schedulable subset, there are a number of criteria that we can use to pick this subset. One possibility is to start with the message with highest priority and keep picking messages with lower priority till no more messages can be included. This is easily implemented as a greedy algorithm and we take this approach in our implementation. Another criteria could be that we pick the largest subset of messages that are schedulable. This could possibly be implemented as a dynamic programming problem. We leave open the possibility of including these alternatives in our implementation.

2.2 Compacting Schedules

If we run the generating algorithm on different configurations, it will result in a automaton with a large number of states. As an example, consider the generated schedule in Figure 2 below. On the left is the schedule with configurations c_1 and c_2. Notice that the two configurations share the first node and also the two terminal nodes. Therefore, it is possible to compact the schedule to generate the schedule on the right.

If we merge the common prefix and suffixes of different configurations, we would get a schedule that is compact without introducing extra guards. It is possible to compact this further by compacting the nodes in between also.

However, notice that compacting the middle nodes means that we need to distinguish between the subsequent paths. Otherwise we would have introduced behaviors that are not consistent with the original schedule. Therefore, to achieve maximum compaction, we would have to introduce additional guard conditions in the schedule that would only permit the original behavior of the system. The compaction in schedule beyond the prefix and suffixes therefore comes at the cost of evaluating guards at runtime. In general, the more compact a schedule, the higher is the computational overhead of evaluating the guards at runtime. The two metrics that can be used to measure the overhead of any compaction are therefore, guard computation time and schedule storage.

We present below an efficient algorithm for generating the most compact schedule - one that consists of storing each message just once.

DFS(σ)

1: **for** each $u \in \sigma$ **do**
2: $color[u] \leftarrow WHITE$
3: $makeLink(u, NULL, NULL)$
4: **end for**
5: **for** each $u \in \sigma$ **do**
6: **if** $color[u] = WHITE$ **then**
7: $DFS - VISIT(u)$
8: **end if**
9: **end for**

DFS-VISIT(u)

1: $color[u] \leftarrow GRAY$
2: **if** $isPresent(u, getInterval(u), hashTable) = FALSE$ **then**
3: $insertInTable(u, getInterval(u), hashTable)$
4: **end if**
5: **for** each $v \in getChildren(u)$ **do**
6: **if** $color[v] = WHITE$ **then**
7: $makeLink(u, v, generateTransitionCondition(u, v))$
8: $DFS - VISIT(v)$
9: **end if**
10: **end for**
11: $color[u] \leftarrow BLACK$

Pseudocode 2: Stateful Schedule Compaction

The Pseudocode 2 is essentially a depth-first traversal of the stateful communication schedule. At each node, we first check to see if we have seen the node instance before. This is accomplished by hashing the node on the basis of the message identifier and the duration of the slot. The method $isPresent$ checks to see if the node is present and the method $getInterval$ returns the interval duration of the slot. We then traverse all the children nodes while making links to the parent. The method $generateTransitionCondition$ generates the transition condition between the parent and the child node. The guard condition essentially involves noting the configuration and the time of arrival at the node. $makeLink$ method is responsible for actually implementing the linking of the nodes.

As an example, consider the schedule in Figure 2. In the schedule on the right, there are still repeated slots for a in configuration c_1 and for node b in configuration c_2. The algorithm above will compact them and produce the compact schedule shown in Figure 3.

Complexity Since the Algorithm 2 stores each unique message instance (messages with unique identifier and slot duration), it is the most optimal in terms of the storage space required to store the schedule. However, there are extra guard conditions that need to be evaluated. For instance, in Figure 3, the extra conditions $t = 3$ are introduced on node a/b that enables the transition if the current discrete time is 3. Therefore, we need to store

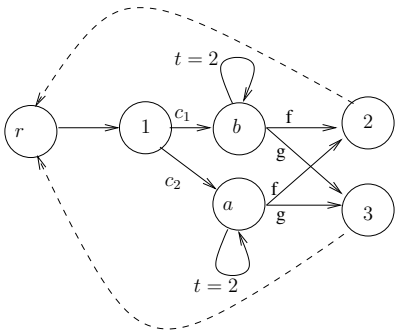

Figure 3. Example of generated compact schedule

these guard conditions as well. The generation algorithm 2 is essentially a depth-first traversal of the original schedule followed by lookups in a hash-table. If we use the universal hashing function on a large prime, we could get minimal collisions and get a constant expected time lookup [5]. Therefore, the whole algorithm runs in *expected* linear time. Note that we use linear with respect to the input, which means the size of the input graph of the schedule. So if the input schedule is of the order of $O(V + E)$, then the algorithm is expected to be of the same order as well.

3 Representation of Stateful Communication Schedules

In this section, we describe our implementation of the generation and compacting techniques introduced above. From the input specification, we generate the compact output schedules as a data structure (text) and as a labeled graph with nodes representing a message being scheduled.

3.1 Implementation

For our implementation, we take the input as a CSV format of all the configurations and messages. We then parse this file and generate the schedule by running it till the LCM of all messages in each configuration. We then compact the generated schedule as discussed in the previous section. In the implementation, we have only considered the earliest-deadline first scheduling strategy. However, we expect that extending it other schemes should be fairly straightforward. All these methods are implemented in perl [2]. We use the graph rendering software graphviz [1] to visualize the schedule. The entire tool chain is described in the Figure 4 below.

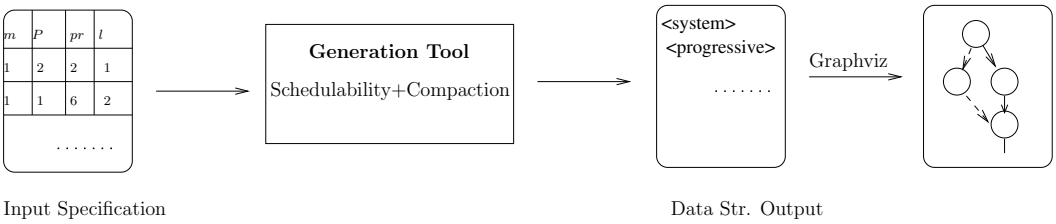

Figure 4. Toolchain for generation and representation of Stateful Communication Schedules.

To run the toolchain, we run the script $treeGenSc$ provided in the implementation via the command line as:
`sh treeGenSc.sh inputFile outputFile`
In case the set of messages is not schedulable, an error message is generated. Figure 5 presents the compact schedule generated for the example set of configurations in Table 2. In the figure, the nodes represent messages,

and the guards (labeled edges in the visualization) are of the form c_i, j represents configuration c_i and iteration j. It should be noted that this is just one of the several possible encodings of guard conditions.

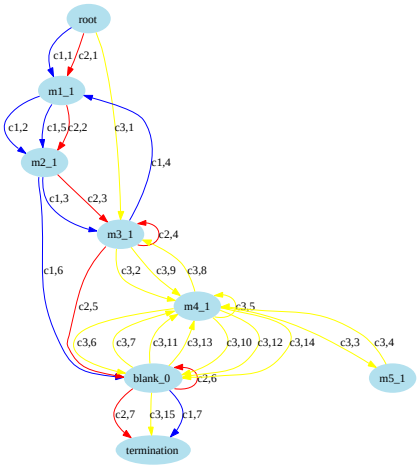

Figure 5. Compact schedule generated for the set of configurations in Table 2.

If we modify the example so that, $c'_1 = c_2$, $c'_2 = c_3$ and c'_3 is same as the first configuration c_1 except that the message m_1 has a period of 2 and m_2 has a period of 3, then the configuration c_3 is no longer schedulable. In this case, an error is thrown and the schedule corresponding to the remaining two configurations is generated. The generated compact schedule is shown in Figure 6.

3.2 Conclusions and Future Work

In this project, we have delved on the problem of generating a stateful communication schedule given an input specification. Starting with information about configurations which contain the frequency and duration of different messages, we generate a stateful schedule in the form of a labeled directed graph that is compact. We have also discussed the generation, schedulability, and compaction algorithms with the associated tradeoffs. We have also described the tool chain that we have developed to generate both a data structure and visual representation of the final schedule.

There are several possibilities for future work. Firstly, we hope to extend the schedulability checks to deadline monotonic scheduling. We are also working on constructing the largest schedulable subset of messages, should the original configuration be not schedulable. Other possible extensions include tackling the problem of composing several stateful schedules using an appropriate composition technique, integrating the analysis framework into the tool chain, and automatic generation of network-code from the stateful communication schedules.

References

[1] Graphviz. `http://www.graphviz.org`.

[2] Perl. `http://www.perl.com`.

[3] M. Anand, S. Fischmeister, and I. Lee. An analysis framework for network-code programs. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 122–131, New York, NY, USA, 2006. ACM Press.

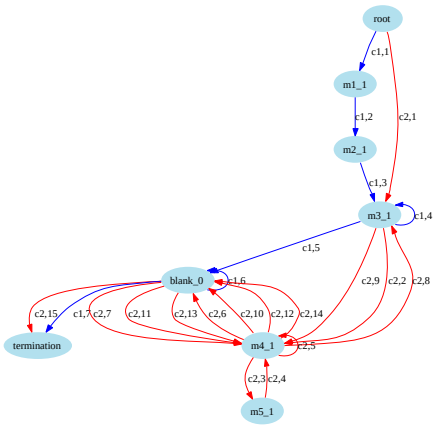

Figure 6. Compact schedule generated when c_3 is no longer schedulable

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

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. 1990.

[6] 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.

[7] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, New York, NY, USA, 2006. ACM Press.

[8] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multihop networks. *IEEE Trans. Parallel Distrib. Syst.*, 5(10):1044–1056, 1994.

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

[10] T. Nghiem, G. J. Pappas, R. Alur, and A. Girard. Time-triggered implementations of dynamic controllers. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 2–11, New York, NY, USA, 2006. ACM Press.

[11] Q. Zheng and K. G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Trans. on Communications*, 42(2/3/4):1096–1105, Feb./Mar./Apr. 1994.

[12] K. M. Zuberi and K. G. Shin. Non-preemptive scheduling of messages on controller area network for real-time control applications. In *RTAS '95: Proceedings of the Real-Time Technology and Applications Symposium*, page 240, Washington, DC, USA, 1995. IEEE Computer Society.