Embedded Network Protocols for Mobile Devices

Despo Galataki[1,2], Andrei Radulescu[2], Kees Verstoep[1], and Wan Fokkink[1]

[1] VU University, Dept. Computer Science, Amsterdam, The Netherlands
[2] ST-Ericsson, Eindhoven, The Netherlands

Abstract. Embedded networks for chip-to-chip networks are emerging as communication infrastructure in mobile devices. We present three novel embedded network protocols: a sliding window protocol, a protocol for opening and closing connections, and a bandwidth reservation protocol. The design of these protocols is tailored to the low power and low cost requirements of mobile devices. The model checker SPIN played an important role in the design and analysis of these protocols. Large instances of the protocols could be analyzed successfully using the distributed model checker DiVinE.

1 Introduction

For certain (e.g., mobile) applications there is too little physical space on the chip packages to accommodate all the necessary traditionally-parallel interfaces. Therefore, there is a shift from parallel interfaces towards high-speed serial interfaces. This trend is visible in, e.g., computer chips [12,16,18], FPGA chips [1,28], and mobile device chips [23,15].

High-speed serial links, while very efficient in terms of energy per bit, have transmission errors which need to be resolved by the protocols above. Moreover, these links are intrinsically point-to-point, which implies that if multiple devices need to be connected together, a network topology must be used.

The trade-offs for designing a chip-to-chip network are different from computer networks [7,22], which are often designed for scalability and throughput, or on-chip networks [4,6,13], which tend to be designed for low cost and power, but have a much higher throughput due to wires being relatively inexpensive. A chip-to-chip network is also designed for low cost and power. Moreover, it must cope with relatively large latencies caused by the transmission serialization, which puts pressure on buffering, one of the most important cost factors. Chip-to-chip interconnects are thus typically designed to offer reliable, in-order communication at the Data Link layer. Additionally, due to the small-scale and controlled environment, and to avoid retransmission buffers at the Transport layer, routers do not drop data when their buffers fill up, but apply backpressure instead.

In computer chip networks, the high-level protocols are memory-based and host-centric to cope with the existing legacy [12,16,18]. In mobile devices, a different approach has been taken, in which, due to the trend towards multi-host sys-

tems, the chip-to-chip networks are emerging as flat and non-hierarchical, offering services similar to those in computer networks, such as TCP-like connection-oriented communication [8,15,21]. A connection-oriented service involves the ability to open connections, which are then used to transfer data, and close connections, such that ports can be reused by the same application to communicate to other nodes, or by a different application. Another aspect when designing chip-to-chip networks for mobile devices is native support for bandwidth reservation to enable correctness by a composable system design [8,21]. This is similar to some approaches for on-chip networks [10,14]. However, instead of a tightly coupled system-wide time-division-multiplexing approach, which is less suitable in an intrinsically asynchronous network, bandwidth is assumed to be allocated at each link. Consequently, it needs to be allocated and deallocated as part of the connection opening and closing stages.

We report on the design and analysis of three core protocols for communication and connection management. We focus on these protocols because their design had to be tailored to the low power and low cost requirements, and model checking played an important role in the design process. We first present a sliding window protocol for the Data Link layer that has been optimized for the target domain. We then present a protocol for opening and closing connections, which takes advantage of in-order delivery in chip-to-chip networks within mobile devices. As a result, the protocol does not use sequence numbers and maximum segment lifetime as in TCP [19]. Finally, we discuss an extension of this connection management protocol that includes in-band link-bandwidth reservations. Due to space restrictions, we cannot explain the protocols in full detail. The reader is referred to [9] for detailed descriptions of the protocols.

During their design, the protocols were analyzed using the SPIN model checker, as well as with DiVinE, which distributes the workload of a verification among multiple compute nodes. DiVinE could verify larger problems than SPIN, while SPIN's detailed error trails were used to find flaws in a particular design and correct them. The use of model checking was crucial in the protocol design. Notably, through verification we learned that an extra phase is needed for the connection management protocol, in contrast to TCP's three-phase connection protocol. Additionally, verification guided us in the design of an optimization of bandwidth (de)allocation to reduce memory overhead.

The protocols were developed in the context of UniPro^{SM}, a serial high-speed interface for interconnecting integrated circuits in mobile phones; it is bound to become part of millions of mobile phones world-wide. It should be noted that UniPro^{SM} is still under development, and the protocols described in this paper will undoubtedly be adapted and extended in the near future, or be replaced by alternative designs, to meet the requirements of the different industrial partners.

2 UniPro^{SM}

The diversity and complexity of the development of mobile phones has created a need for standardization, which is addressed by the Mobile Industry Processor

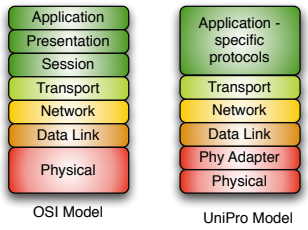

Fig. 1. OSI and UniPro℠ network layers

Interface (MIPI®). MIPI®, which is supported by an alliance of most mobile industry companies, defines the interface standards for mobile phones features, like audio, displays and cameras. In particular, there is a need for a general protocol that is responsible for the communication among applications and devices. This is the responsibility of the UniPro℠ [15] layer stack. UniPro℠ can support networks of up to 128 devices (integrated circuits, camera processor, displays, baseband, etc). It is a generic hardware- and software-friendly technology, which can support a diversity of applications. UniPro℠ offers low-power modes through the physical layer underneath to minimize power consumption. Other important requirements are low memory consumption, high speed, reliability and robustness, even in the face of failures in mobile devices, message loss and crashing applications. Inspired by the new era of multitasking, UniPro℠ is ready for upcoming innovations of parallel processing on mobile devices as well.

UniPro℠ is largely based on the OSI Reference Model. From Fig. 1, one can observe some differences between the two models. UniPro℠ partitions the physical layer in two. The lowest layer is in charge of electrical signaling, line encoding, etc. (like the physical layer in the OSI model), while the intermediate layer (Phy Adapter) is responsible for abstracting the different technologies and combining them in a heterogeneous environment. The Data Link layer ensures that there is a reliable link between two modules in one hop distance, and that a frame can be arbitrated and multiplexed corresponding to the specified priorities. Similar to OSI, the Network layer deals with routing and addressing packets. The Transport layer defines the quality of a connection and is responsible for the flow and congestion control of the network. The UniPro℠ model combines the three upper layers of the OSI model – Session, Presentation and Application – into a single one, because it is responsible for connecting the diversity of applications and modules together rather than for implementing applications. The interface of the Transport layer has to be simple, so that applications can be easily adapted to it.

3 Sliding Window Protocol

Errors may occur on the links and routers may get overflowed, so messages can get lost. As a result, a continuous flow of communication between a sender and a

receiver (data packets and acknowledgments providing feedback that they have been received) has to be established by dedicated protocols. Sliding window protocols (see, e.g., [22]) offer reliable data transmission and control the flow of messages, accommodating differences in link and processing speeds. Sliding window variations are used at both the Data Link layer (HDLC) and in the Transport layer (TCP) of the OSI model.

The data being transferred from a sender to a receiver is fragmented into packets. The packets carry sequence numbers, which can be seen as a running index into the buffered packets at the sender, with an extra bit to avoid confusion between old and new fragments. The receiver sends as acknowledgment (ACK) the sequence number of a received packet to the sender. It may also send a negative acknowledgment (NAC) in case of a failure. Sliding window protocols are typically enhanced with optimizations, e.g., to hide latency of transmission and increase the network utilization by pipelining techniques. An example is TCP [5], which in addition uses the maximum packet lifetime and an estimate of the round trip time [24]. Variations of sliding window protocols have been studied and formally verified in different ways (see, e.g., the related work section in [2]).

There are two generic sliding window protocols in the literature [22]. One version, called *go-back-N*, is that the receiver ignores all packets after an error until it receives the correct one; the sender resends *all* packets that have not been acknowledged, after a timeout. The second version, called *selective repeat*, is that only failed packets are resent; the receiver informs the sender if there is a failure and on which packet. Go-back-N wastes time, compared to selective repeat, because the sender needs a timeout to learn about failed packets. On the other hand, it is simpler and gives less memory overhead at the receiver. The sliding window protocol (SWP) we developed for the Data Link layer is a mixture of go-back-N and selective repeat. When the receiver notices a failure, as in selective repeat, it sends NAC_i with i the sequence number of the last correct packet it received. The sender thus gets to know about the failure earlier than if it had to wait for a timeout. As in go-back-N, the receiver ignores all packets after an error until it receives the correct one.

The sender's flow chart is shown in Fig. 2. In general, the sender can send up to N packets to the network, and it can only send the next one when some of the packets that it sent are acknowledged by the receiver. It will resend a packet only if it receives a NAC or after a timeout. The sender needs to store any two out of three predicates, *beginning*, *on_post* and *current*. These are the basic variables defining the sender's *window* of packets which have been sent but not acknowledged. By maintaining two of these variables for a connection, the sender can easily derive the third one, because $\mathit{beginning} = \mathit{current} - \mathit{on\_post}$.

- *beginning*: Indicates the first packet that was sent but not acknowledged.
- *on_post*: Indicates the number of the packets waiting for an acknowledgment; it can be no more than the maximum window size.
- *current*: Indicates the next packet that will be sent.

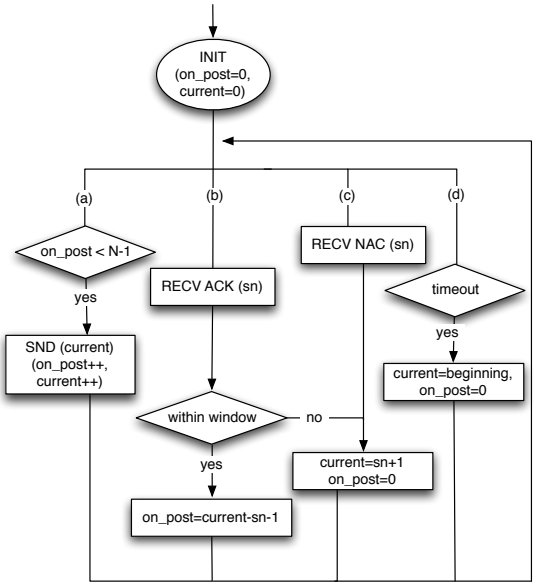

Fig. 2. SWP sender

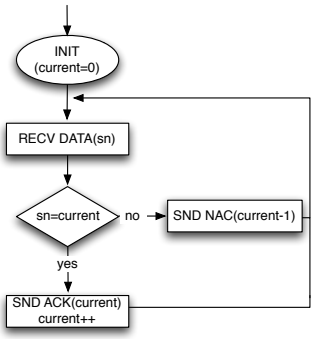

Fig. 3. SWP receiver

Fig. 3 shows a flow chart of the receiver's algorithm. The protocol starts with $current = 0$, where *current* indicates the identity of the expected packet. When the receiver receives a packet, it checks if it is the expected one; if so, it sends back an acknowledgment ACK_i, and waits for the next packet to arrive by incrementing *current*. In case an unexpected or garbled packet arrives, it sends NAC_{i-1} with the identity of the last correct packet which arrived in order.

The resulting protocol has the advantage of little memory overhead (the same as go-back-N), while giving a significant recovery time gain compared to go-back-N. For further details, the reader is referred to [9, Sect. 3].

4 Connection Management Protocol

The connection management protocol (CMP) presented here is based on the well-known TCP connection protocol, with its *three-way handshake*, which works as follows. A client initiates a connection by sending a synchronization request (SYN) to a server. The server, if readily available, acknowledges the request. Finally the client sends an acknowledgment back. The client repeats sending a SYN and the server repeats sending an ACK when a timeout occurs. After reception of a client's ACK, both end nodes are connected and ready to exchange data. If a node wants to leave, it informs the other party by sending a finalization request (FIN), and waits for an acknowledgment. If this acknowledgment is delayed, then after a timeout it resends the FIN. After receiving a FIN, a node can continue to send data until it is also ready to close the connection.

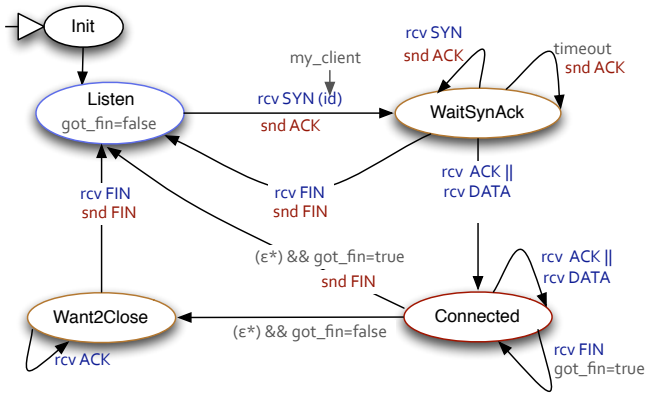

Fig. 4. Server's state machine with its current client

To improve power and memory consumption, we make a number of adaptations to TCP's connection management protocol. We aim at minimizing the number of exchanged messages, memory overhead and completion time. Tables that hold history information and interpretations whether a delayed message has become obsolete are excluded. Session identification of a connection and timing variables are kept to a minimum. Messages may be dropped due to resource contention, if there is a shortage of buffer space or processing power. However, the network is designed to deliver messages in order.

Part of the state machine of a server is displayed in Fig. 4 (for the interplay of a server with a node that is not its client, see [9, Fig. 22]). The initial state is Listen. The received messages are from its client. The states are as follows:

- Listen: The server is free to accept a new connection and is not busy with a client. When it receives a SYN from a new client, it sends back an ACK and proceeds to WaitSynAck.
- WaitSynAck: The server can receive an ACK or a DATA (a message containing data), indicating its client received its ACK and is connected. The server then moves to Connected. If it receives a FIN, it replies with FIN and goes back to Listen. The FIN may indicate that the client does not want to use the connection anymore. The server stays in the same state if it receives another SYN or a timeout; in both cases it sends ACK to its client.
- Connected: The server is participating in a data exchange. It stays in the same state if it receives an ACK, DATA or FIN; they are not answered. At reception of a FIN, it sets $got\_fin = true$; (ε^*) means that the server closes the connection. If the server already received a FIN from its client, it can reply with FIN and move to Listen. Otherwise, it just moves to Want2Close.
- Want2Close: The server can receive an ACK in case there was a repeated and delayed ACK from the client. Then the server stays in the same state. When a FIN arrives, the server answers with FIN and moves to Listen.

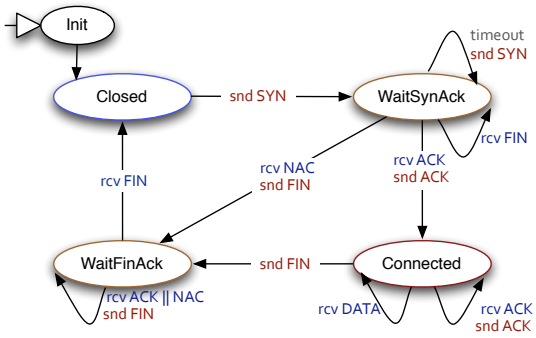

Fig. 5. Client's state machine with its current server

If the server receives a FIN or (except for state Listen) SYN from a node that is not its client, it replies with FIN or NAC, resp., and stays in the same state.

Fig. 5 shows a client's interaction with its server (for the interplay of a client with a node that is not its server, see [9, Fig. 24]). The initial state is Closed.

- Closed: The client chooses a server and tries to connect to it by sending SYN and moving to WaitSynAck.
- WaitSynAck: The client expects to receive an ACK, which it answers with ACK. It may receive a NAC, indicating the server is busy. It is important that in this case, the client replies with FIN and moves to WaitFinAck (this will be explained in detail below). SYN is replayed after a timeout. If the client receives a FIN, this means the server replied to a FIN of an old connection.
- Connected: The client can receive another ACK, after which it sends ACK back to its server. As this is the state where data exchange is done, the client generally receives some DATA too. When it does not want to send more data, it informs the server with a FIN and moves to WaitFinAck. Notice that the client should not receive any FIN from its server before it sends its own FIN.
- WaitFinAck: The client waits for a FIN, after which it goes to Closed. Apart from a successful request (through Connected), the client also reaches this state after it receives a NAC. That means it can receive multiple ACKs and NACs before it gets a FIN from the server.

In every state a FIN can arrive, as a delayed repeated message from a server of an old connection. The client ignores such messages.

We explain why the client should send FIN after the server answers with NAC to a SYN. The other option would be that the client simply stops trying to connect and moves to Closed. The server is not affected, as at the moment it answered to the SYN, it was busy with another client and it did not initiate any new connection. This is a fast and simple way to close the connection. However, by means of the model checker SPIN, we found a flaw in this idea, depicted by the scenario in Fig. 6. The client sends two SYNs to the server. While receiving

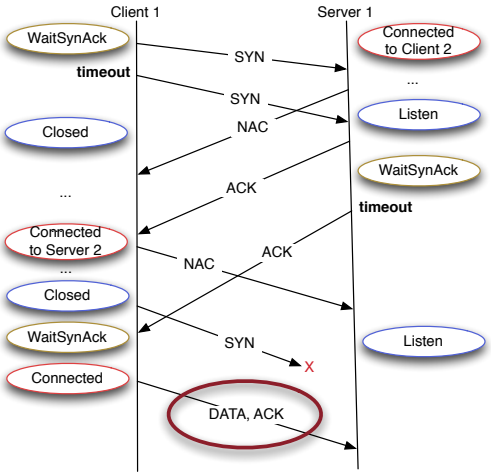

Fig. 6. An example where the client closes immediately after a server's NAC without going through WaitFinAck

the first one, the server is busy with another client, and thus answers with NAC. By the second SYN, the server is ready to set up a new connection and replies with ACK. Then it moves to WaitSynAck, where a timeout occurs, and as a result it replays ACK. In the meantime, the client has received a NAC and has moved to the Closed state. As a result, the client may follow up connecting to another server. Consequently, when the client receives the first ACK from the first server, it responds with NAC. As a result, the server moves to Listen (in the correct version of the protocol this NAC to the server cannot happen, therefore it is not considered in Fig. 4). The client decides to reconnect to the server, and receives an ACK from it. However, the received ACK is from the first connection attempt, and the server is not aware of this new connection, because the corresponding SYN was lost, due to resource contention. The client incorrectly assumes it has a connection, and starts sending data to the server.

If we can distinguish SYN messages from different sessions (replayed SYNs are considered to be in the same session), the problem is solved. A trivial solution is to keep track of the last session of all servers at the client side and all clients at the server side. This solution, however, does not scale. Trying to keep track of all different sessions with only one extra bit is not possible, because servers and clients can connect to each other multiple times.

By asking the client to close the connection via WaitFinAck, we prevent it from connecting to another server until it receives the server's FIN. The main idea is that the client can only move to the next session when it is certain it will not receive any more ACKs and NACs from the server for this session. Once the server sends ACK, it moves to WaitSynAck, and after that, it can only send ACKs after a timeout, until it gets an answer from the client. Hence the client

receives at least one of the NACs or ACKs before it goes to the next session. Once it receives an ACK, it can only receive ACKs until it moves to the next session. If none of the NACs arrive at the client, it sends SYNs until it gets an ACK, and then moves to Connected. If a NAC arrives at the client, it answers ACKs and NACs with a FIN, until it gets a FIN from the server, and then it moves to Closed. Thus we make sure that both ends absorb all the SYNs, ACKs and NACs for this session before moving to the next session.

We have simplified the closing of connections by enforcing that the client is always the first to send a FIN to the server. The server thus always closes before the client. In addition, the client is the one who requests a new connection. Therefore, there is no way to mix one session with another. One could claim that having the client always close first is a limitation of the protocol. This can be hidden, however, as we could give the server (in Want2Close) the option to piggyback a flag that it wants to close the connection.

5 Router Management Protocol

We now turn to the router management protocol (RMP) for congestion avoidance, on top of CMP. The protocol is able to avoid overloading paths in the network by making explicit bandwidth allocations at the routers for every segment of the path. First we sketch how it works when the bandwidth allocation succeeds; see Fig. 7 for an illustration of this procedure. The client starts by sending SYN to the server. To this message it attaches the bandwidth ($bw1$) that needs to be reserved. Routers do not make a reservation on receiving this SYN, but just forward it to the next router or the server. The server sends an ACK with an aggregate value ($bw1+bw2$) of the client's and its own bandwidth.

A router, when receiving an ACK from server to client, first searches if the triple (client, server, bandwidth) already exists in its memory. If not, and if the router has sufficient remaining bandwidth, it creates a triple with bandwidth $bw1 + bw2$, which reserves this bandwidth to the connection. The connection is established by the subsequent ACK from client to server, and then data can be exchanged. The routers wait until they receive a FIN from the server to the client, indicating the end of the connection on both sides. Then the router checks if a corresponding triple exists in memory. If so, it reclaims the bandwidth for this connection and removes the triple from memory.

Suppose a router, when receiving an ACK from server to client, finds it has insufficient remaining bandwidth. As illustrated in Fig. 8, the router then sends ERR in the client's direction without storing the triple. If this ERR gets lost, the client replays SYN or the server replays ACK after a timeout, invoking another ERR at the router. When the client finally receives the ERR, it closes the connection exactly as when it receives a NAC (by sending a FIN and waiting for the server's FIN). When a second ACK arrives at the router, the router may in the meantime have freed adequate bandwidth to serve the connection. Then the connection can still get established, if the ERR never arrived at the client.

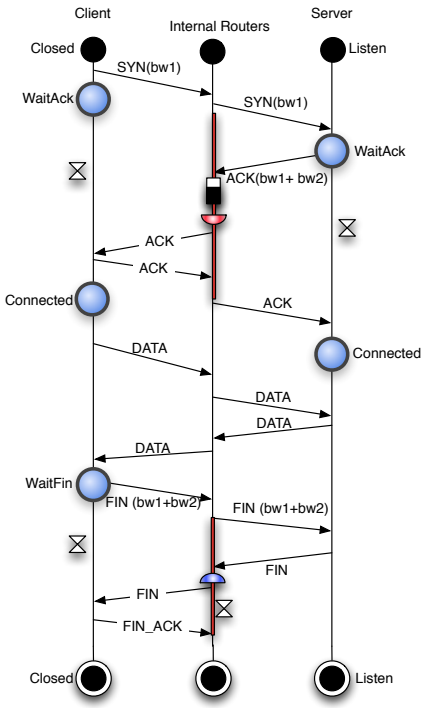

Fig. 7. Bandwidth allocation succeeds

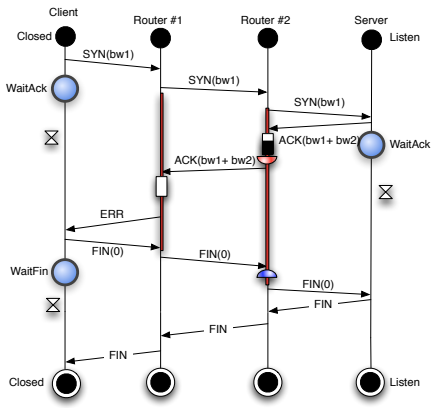

Fig. 8. Bandwidth allocation fails

When a connection is being closed, routers must be able to distinguish the first FIN received from a client and repeated FINs for closing this same connection. Otherwise routers could reclaim bandwidth for the same closing connection multiple times. An easy solution is to let routers store triples (client, server, bandwidth) until the end of a connection. However, since there can be hundreds of active connections, this imposes a relatively heavy memory load. Therefore, in the final version of the protocol we introduced an optimization in which such triples are only kept in the routers' memory while setting up and closing the corresponding connection, and not during data exchange.

When a router receives an ACK from client to server, it looks if there is a corresponding triple in its memory, and if so, removes this triple. On the other hand, when a client closes a connection, it attaches the bandwidth of this connection to FIN, so that routers can restore the triple. To remove this triple from the routers' memory again, we add an extra message at the very end of CMP. After a client has received a FIN from its server, finalizing the closure of the connection, it sends one extra message back, to inform intermediate routers that they can remove the corresponding triple.

6 Model Checking Analysis

We applied the model checker SPIN [11] during the design of the SWP, CMP and RMP protocols. SPIN is widely used to analyze real-life communication protocols. The tool can discover potential deadlocks, livelocks or invalid states. In addition, properties written in LTL (Linear Time Logic) can be checked.

SPIN has a wide range of analysis options, e.g., live simulation, and full scale or approximate state space analysis. An important characteristic of SPIN specifications are the dimensions of the various model state variables. These maximum dimensions need to be chosen carefully, or the corresponding state space will very quickly grow such that full scale analysis is no longer feasible.

To reduce the relevant state space to a managable size, a wide range of techniques is reported on in the literature [20], but applying them successfully may require significant expertise and often some amount of experimentation while "tuning" the model. As a result, memory requirements are frequently the bottleneck in being able to analyze larger protocol instances. It can thus be beneficial to employ analysis tools using a large distributed memory, provided that both data and computation can be distributed effectively.

A prominent example in the category of distributed LTL model checkers is the DiVinE [3] system. As shown in [25], the DiVinE model checker has good scalability on clusters with a fast interconnect, but can also be applied successfully in a high-bandwidth computational grid environment. Unlike sequential model checkers, which typically use depth-first search, DiVinE uses breadth-first search (which parallellizes well) and employs a hashing function to evenly spread the state space and work load over the compute nodes. To facilitate LTL model checking, which requires a cycle detection algorithm, DiVinE implements various distributed algorithms. In this paper we used the "OWCTY" algorithm, which is based on a distributed version of Topological Sort.

DiVinE supports both a native modeling language "DVE" and codes written in SPIN's modeling language Promela. Promela specifications are handled by DiVinE using the embedded "NIPS" module. NIPS is a complete reimplementation of the original SPIN tool, by means of a specially developed model-checking *virtual machine* [27]. An interesting aspect of this SPIN reimplementation effort is that the resulting model checking byte code can be optimized off-line by additional tools, which can significantly reduce the resulting state space. Practical examples of these reductions will be discussed below. Instead of using Promela, the protocols discussed might also have been modeled in DVE, giving an additional performance gain. However, for pragmatic reasons we chose Promela.

We ran DiVinE on 64 compute nodes of the DAS-3 cluster (`www.cs.vu.nl/das3/`) at VU University. The 2.4 GHz AMD Opteron-based nodes are interconnected by a fast Myri-10G network, and have 4 Gigabyte of memory each.

6.1 Model Checking SWP

For SWP, we checked a number of LTL properties that together assert the required behavior of the protocol, i.e., it should eventually deliver all messages,

Window	SPIN states	DiVinE/NIPS states				
		PR/DVR/SCR	PR	DVR	SCR	Base
2	$1.38 * 10^5$	$1.00 * 10^5$	$1.56 * 10^5$	$1.47 * 10^6$	$1.75 * 10^6$	$1.75 * 10^6$
3	$3.19 * 10^6$	$2.00 * 10^6$	$3.62 * 10^6$	$3.55 * 10^7$	$4.44 * 10^7$	$4.44 * 10^7$
4	$5.11 * 10^7$	$2.95 * 10^7$	$5.78 * 10^7$	$5.82 * 10^8$	$7.43 * 10^8$	$7.43 * 10^8$

Table 1. States in the SWP LTL=1 for SPIN and DiVinE/NIPS

in order, without duplication, despite possibly losing packets. In particular, we looked at the following LTL properties:

- LTL 1: no message is duplicated
- LTL 2: messages are not reordered
- LTL 3: every data message sent is eventually received

We also included LTL 4, which is a combination of LTL 2 and 3, and LTL 5, which is an alternative formulation of LTL 3. We will focus on LTL 1 and 5, being representative for the model checking effort required (e.g., the size of the resulting state space). For the formulation of these properties, see [9].

With sequential SPIN, we could indeed check all properties. However, the state space growth when gradually increasing the maximum window size (the most important model parameter) was considerable. As illustrated in Table 1, the growth rate is over an order of magnitude for every increment of the window size. As a result, analyzing the properties for a window size of 4 is already becoming difficult, as the state space exceeds available memory (we determined the largest state space on a special DAS-3 node equipped with more memory than the 4 GByte available by default). By enabling SPIN's state compression methods, the state space capacity can be extended, but the most efficient compression technique comes at a significant runtime cost – potentially further increasing the high runtime by a factor of ten of more. It is worth mentioning that the SWP specification discussed is already optimized using most well-known SPIN state space reduction techniques available; unoptimized initial versions of the SWP specification could in fact only be analyzed up to a window size of 2. On the other hand, a full-scale analysis for a window size of 4 appears reasonable, given the target setting.

Table 1 also shows the sizes of the state spaces using the SPIN support in DiVinE. The unoptimized NIPS bytecode (the "Base" version in the table) induces a much larger state space than SPIN. However, successive bytecode optimizations by means of the SARN [17] toolset reduce the effective state space to somewhat below the state space reported by SPIN with its default partial order optimization enabled. The SARN tools applied are Path Reduction (PR), Dead Variable Reduction (DVR) and Step Confluence Reduction (SCR). PR appears to be the optimization with the largest impact, since it most effectively reduces the number of synchronization points in the model checking byte code.

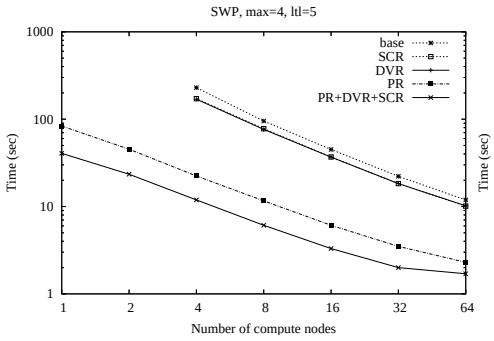

Fig. 9. SWP state space reduction impact on DiVinE runtime

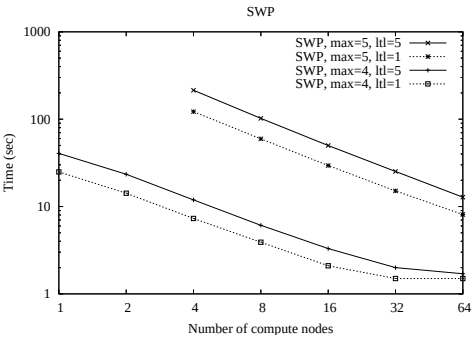

Fig. 10. SWP model scaling and LTL impact on DiVinE runtime

Problem	Instance	SPIN 1 node	DiVinE 1 node	DiVinE 4 nodes	DiVinE 16 nodes	DiVinE 64 nodes
SWP	max=4,ltl=1	8.8	24.9	7.3	2.1	1.5
	max=4,ltl=5	18.2	40.6	11.9	3.3	1.7
	max=5,ltl=1	158	448	122	29.5	8.1
	max=5,ltl=5	324	819	214	49.9	12.8

Table 2. SPIN and DiVinE SWP run times in sec. for large instances (note that state space volumes are not identical).

Fig. 9 shows the effects of state space reduction on the DiVinE running time. Note that the figure is log-log scaled to account for the wide range in state spaces (due to static optimization discussed above) and parallel running times. The figure indicates that DiVinE is able to achieve almost linear speedup up to 32 compute nodes, and for the larger problem sizes up to 64 compute nodes. A similar pattern can be seen in Fig. 10, where the state space variation is induced by scalings in the maximum window size (and the LTL formula). Finally, Table 2 shows the running times of SPIN and DiVinE, the latter on 1, 4, 16 and 64 nodes.

6.2 Model Checking CMP

For the analysis of CMP, the state machines for the client and server shown in Sect. 4 were transformed into Promela code. Assertions were added regarding messages that should be impossible to be received in particular states. The protocol was instantiated with a configuration of two clients making a sequence of arbitrary connections to two servers. For the initial analysis we used SPIN in default mode, i.e., checking for possible deadlocks, unreachable code, invalid

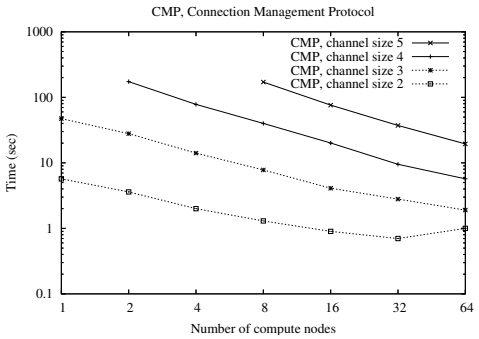

Fig. 11. CMP analysis time using DiVinE

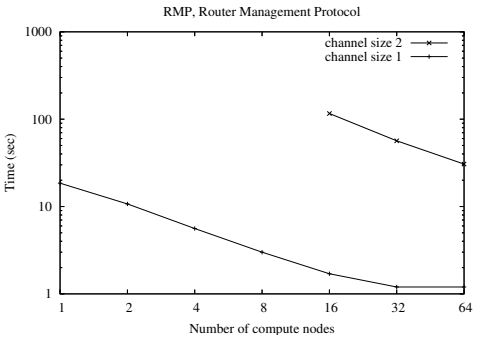

Fig. 12. RMP analysis time using DiVinE

Problem	Instance	SPIN 1 node	DiVinE 1 node	DiVinE 4 nodes	DiVinE 16 nodes	DiVinE 64 nodes
CMP	cap=2	103	5.7	2.0	0.9	1.0
	cap=3	N/A	47.3	14.1	4.1	1.9
	cap=4	N/A	291.7	78.1	20.1	5.7
RMP	cap=1	18.4	18.5	5.6	1.7	1.2
	cap=2	1470	N/A	N/A	116.2	30.6

Table 3. SPIN and DiVinE run times for CMP and RMP in sec. for large instances (note that state space volumes are not identical).

end states and assertions. As explained in Sect. 4, this SPIN analysis led to the detection of a flaw in our original CMP, where the server answered with NAC to a SYN. The rest of this section discusses the analysis of the corrected CMP.

As an additional CMP model parameter we varied the capacity of the channels between the client and server processes. Asynchronous communication with the channel size set to one found no errors, but detected some unreachable code, which indicates that some valid scenarios may not have been analyzed with a channel size this small. Parallel performance of the DiVinE analysis of CMP using channel capacity between two and five is shown in Fig. 11.

Table 3 compares the running times of SPIN and DiVinE. The entries marked N/A could not be completed due to memory shortage. The state space corresponding to the CMP protocol is again very effectively reduced by the SARN toolset, in particular by its Step Confluence Reduction (SCR) tool. By merging equivalent sets of states based on program location, SCR here reduces the state space almost by a factor of 60, allowing DiVinE with NIPS and SARN to outperform SPIN even on a single compute node, which is rather uncommon.

6.3 Model Checking RMP

For the analysis of RMP, the model of CMP was extended with explicit routing nodes between a client and server. A fixed configuration of three routing nodes was used to represent arbitrary setups involving an initial, intermediate, and final routers. State regarding remaining bandwidth described by the (client, server, bandwidth) triples was modeled for every router explicitly, and referred to in assertions for particular states. We used an LTL expression to verify that the router bandwidth allocation does not exceed capacity (no duplicated bandwidth allocation) and does not become negative (no duplicated bandwidth releases).

Parallel performance of a DiVinE analysis of RMP is shown in Fig. 12. As the figure shows, RMP displays quite extreme effects on the state space when the model parameter for the channel capacity is scaled up, making a distributed analysis with DiVinE attractive.

7 Conclusions

In this paper we discussed the design of three embedded networking protocols that were tailored to the specific resource requirements of novel mobile devices. We investigated a sliding window protocol, a protocol for connection establishment and a related bandwidth reservation protocol. In designing the protocols, the SPIN model checking tool was very helpful in preventing errors in the protocol descriptions at a very early stage. This should be contrasted with scenarios where a design is already mostly pinned down or an actual implementation exists, which first has to be reformulated back into a different modeling language.

In the models we checked deadlock freeness, various assertions on states, as well as more general properties formulated in LTL. As the protocol designs became more mature, the checking of larger model instances was attempted. The state space explosion phenomenon forced us to apply a range of SPIN model "optimization" techniques to significantly reduce the effective protocol state space. Unfortunately, this forces a modeler to focus on low-level SPIN implementation aspects which are mostly irrelevant to the abstract model as such.

Despite extensive state space reductions achieved on the models, several realistic instances still were infeasible for analysis with SPIN, due to the limited memory capacity. These larger instances were then checked with the distributed DiVinE tool, which also supports SPIN specifications. This should be weighed against the limited support for error tracing in the SPIN version of DiVinE; for effective work with Promela specifications, use of SPIN itself is currently indispensible. Though DiVinE sequentially runs slower than the highly optimized SPIN tool, given a fast cluster network it exhibits excellent scalability on large problems, making it a useful option for cluster environments with a large distributed memory capacity.

An additional advantage of large-scale distributed model checking is that it can make an approach where model checking is applied to the *target* application language (e.g., as in Java PathFinder [26]) able to efficiently deal with realistic instances, despite the larger state space.

References

1. Altera Corp. and Innocor Ltd. *SerialLite Protocol Specification*, 2003. Rev. 1.0.
2. B. Badban, W. J. Fokkink, and J. C. van de Pol. Mechanical Verification of a Two-way Sliding Window Protocol. In *Proc. CPA*, pages 179–202. IOS Press, 2008.
3. J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *Proc. FMCO*, volume 4590 of *LNCS*, pages 281–293. Springer, 2006.
4. L. Benini and G. De Micheli. *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.
5. D. E. Comer. *Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architecture*. Prentice Hall, 2006.
6. W. J. Dally and B. Towles. Route Packets, Net Wires: On-Chip Interconnection Networks. In *Proc. DAC*, pages 684–689. ACM, 2001.
7. W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.
8. G. Desoli and E. Filippi. An Outlook on the Evolution of Mobile Terminals: From Monolithic to Modular Multiradio, Multiapplication Platforms. *IEEE Circuits and Systems Magazine*, 6(2):17–29, 2006.
9. D. Galataki. Design and Analysis of UniPro Protocols for Mobile Phones. Master's thesis, VU University Amsterdam, 2009.
10. K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.
11. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
12. HyperTransport® Technology Consortium. *HyperTransport I/O Link Specification*, 2009. Revision 3.10b.
13. A. Jantsch and H. Tenhunen, editors. *Networks on Chip*. Kluwer, 2003.
14. S. M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip. In *Proc. DATE*, pages 890–895. IEEE, 2004.
15. MIPI® Alliance. *Specification for Unified Protocol (*UniPro℠*)*, 2010. Version 1.10.
16. PCI-SIG. *PCI Express® Base Specification*, 2009. Revision 2.1.
17. G. Quirós Araya. Static Byte-Code Analysis for State Space Reduction. Master's thesis, RWTH University, Aachen, 2006.
18. RapidIO Trade Association. *RapidIO® Interconnect Specfication*, 2009. Rev. 2.1.
19. RFC 793: Transmission Control Protocol, 1981. Edited by Jon Postel.
20. T. C. Ruys. Low-Fat Recipes for SPIN. In *Proc. SPIN*, volume 1885 of *LNCS*, pages 287–321. Springer, 2000.
21. R. Suoranta. New Directions in Mobile Device Architectures. In *Proc. DSD*, pages 17–26. IEEE, 2006.
22. A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 2002.
23. MIPI® Alliance. *Specification for D-PHY*, 2009. Version 1.00.00.
24. A. Udaya Shankar. Verified Data Transfer Protocols with Variable Flow Control. *ACM Transactions on Computer Systems*, 7(3):281–316, 1989.
25. K. Verstoep, H. E. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *Proc. IPDPS*. IEEE, 2009.
26. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
27. M. Weber. An Embeddable Virtual Machine for State Space Generation. In *Proc. SPIN*, volume 4595 of *LNCS*, pages 168–186. Springer, 2007.
28. XILINX®. *Aurora Protocol Specification*, 2003. SP002 (v1.2).