Model Extraction for Sockets-based Distributed Programs

Alexander Heußner

LaBRI, Université Bordeaux – France
ANR AVeriSS

Abstract. The following short report introduces an approach to utilize unbounded, reliable fifo queues as basis for an operational semantics of distributed applications whose implementation is based on the Berkeley Sockets Api. After a sketch of Tcp and the Sockets Api, a first draft of a formalization of distributed programs is given as well as a straight-forward translation into the abstract formal model of queueing concurrent processes.

1 Introduction

Distributed programs that communicate by point-to-point connections over a Tcp/Ip-based network architecture are an important and ubiquitous class of applications whose complexity demands for automatic verification tools to guarantee the communication protocol's correctness. A secure protocol demands a safe transportation of messages, and, hence, most often relies on an underlying layer of Tcp whereas the interprocess communication is implemented with the help of the Berkeley Sockets Api (or "Sockets" for short). Standard examples for these kind of distributed programs are Internet clients and servers, as well as peer-to-peer and Grid computing applications. Classical verification questions are, for example, safety, boundedness, and deadlock freeness [Hol91]. Special attention has to be paid to partial verification or verification with respect to a given protocol specification, as the communicating programs are developed apart and independently, viz. the gargantuan number of Ftp clients and servers which – in practice – should work flawlessly together.

The Berkeley Sockets Api [BSD83] became the de facto standard for applications that communicate via Internet sockets, and most present-day operating systems contain an implementation of this Api (e.g., WinSock on the Windows platform, or as part of the glibc on Gnu/Linux). Even though originally implemented in and for C, almost all modern socket Apis in other programming languages follow its interface structure. In general, Sockets supports a wide variety of different protocol families. In this short report, we will restrict our discussion to the transmission control protocol (Tcp).

For example, Internet servers focus reliability and safety, which requires complex control structures and additional code that assures the proper behavior in (almost) all execution contexts (including also rather unlikely conditions). The

automatic verification of legacy code cannot rely on building formal abstract models by hand from the source code as this would be a tedious task regarding the shear size and complexity of a typical server program. Hence, verification requires a technique for *model extraction*. An obvious demand for the formal model derived from code is the retransfer of the results of formal verification on this model to the original program. This depends on proving the underlying translation to be "semantically equivalent", e.g., a (bi-)simulation, or an over-/under-approximation with respect to the property to verify.

An additional problem with Sockets is the missing formal semantics: the behavior of Api calls is fixed – at least to our knowledge – only by informal, natural language documents like [Ste04] as well as by its implementations on different platforms which differ significantly (see [Ste04] for details). Consequently, justifying a model extraction method would additionally demand a formally complete and correct rigorous semantics for Sockets.

Our Contribution

We present a semi-formalization of distributed programs whose communication is implemented via calls to the Tcp part of Sockets (under certain additional background restrictions). We propose queueing concurrent processes (Qcp) as formal model to extract the basic features of distributed programs. Qcp consists of local automata models (finite automata, counter automata, pushdown automata, etc.) that communicate via reliable, a priori unbounded, point-to-point fifo channels. We conclude by a discussion why these kind of fifo channels are semantically appropriate albeit our background restrictions.

Related Work

A Sockets semantics based on *bounded* fifo channels and thereupon the application of the Spin model checker for verification of Sockets based applications was presented in [CMGMS05]. This publication proposes a formal semantics for Sockets, before it shows the translation to Promela. This approach presents only the Sockets Api calls symbolically, whereas to surrounding code is simulated by Spin's feature to inline C code directly (which poses a problem regarding the symbolic properties one can verify against).

The previous discussion originated from a more general approach to utilize Spin for the verification of event driven distributive reactive systems [HS02], for example, to verify the alternating bit protocol.

A model for (unicast) Udp Sockets based, distributed programs that includes the handling of Icmp error messages was presented in [WNSS02]; the latter also introduces a formal semantic model of Sockets, and relies on the Hol theorem prover for programs written in a subset of OCaml. A general comparison of different angles to attack Sockets' semantic vagueness is presented in [BFN+05].

2 TCP

The *transmission control protocol* (TCP) [RFC793] is the original, and nowadays omnipresent transport layer protocol of the Internet protocol suite TCP/IP [Ste95]. TCP is connection oriented, (per default) full-duplex, stateful (i.e., a connection is closed, half-open, open, half-closed, closed), and assures the reliable transmission of information. A connection binds two different TCP-sockets which both are given by an IP-number (32 bit unsigned) and a TCP-port number (16 bit unsigned). The operating system assigns sockets to running processes dynamically such that at each time each socket is bound to no more than one process. We will call the two processes linked by a connection *peers*.

Let a *message* be the atomic unit of transferred data, e.g., a letter over a finite alphabet. Note that TCP does not transfer messages but (finite) streams of packages; TCP collects certain smaller message to one package (i.e., TCP's atomic unit of transmission) or divides larger messages into smaller ones depending on the byte-encoding of messages. Nevertheless, TCP assures that a sequence of messages sent by one of the two peers will be received in the same order and without transmission error by the other. Hence, an application based on TCP does not need to assure the integrity of the sent data itself (in contrast to UDP based programs which also need to handle ICMP error messages).

An integral part of TCP is the sliding window protocol that determines the maximal number of packages currently in transit. Again, this is negotiated when a connection is established; the maximal size of data currently in transit is $2^{16} - 1$ (in bytes) in the standard case or $2^{30} - 1$ when considering TCP window scaling [RFC1323] (which is enabled per default in most modern operating systems). Note that the latter amounts to $1GB$ of data currently on its way on the network, and that this $1GB$ can be only one continuous message or a gargantuan number of one bit messages. Nevertheless, the actual window size is dynamically adapted by the protocol itself to current network conditions (which depend on *all* participating nodes and routers in the network). Consequently, we need consider alternative models for this non-deterministic behavior: one can either overapproximate the window size by the maximal number of messages in transit at the same moment, or assume the window to be of "unbounded" size. We prefer the latter as it includes non-deterministically all possible window sizes, even if we possibly deal with window sizes that are in practice impossible (with respect to [RFC1323]).

Besides these standard behavior, TCP includes numerous advanced features which need to be introduced here, as server programs tend to take any advantage possible to gain a better performance and higher reliability. Two important features are the direct access to its input and output buffer as well as out-of-band data. Each peer assigned to a socket has an input and an output buffer, these are bounded (fifo) buffers whereas the size is fixed when establishing the connection (there is a maximal size as an operating system variable); nevertheless, each peer can deliberately change unsent data in the output buffer (which is rarely done), or can peek ahead in the input buffer. All the messages that are currently transferred by the network can be seen as being stored in an unbounded fifo

buffer which results in Figure 2. Out-of-band data transmission allows to transfer a signal from one peer to the other “quickly” and independent of all the other data currently in transfer. Most TCP implementations do not establish an extra connection but tag TCP messages by an “urgent” flag while sending, such that it will be delivered immediately to the receiving peer and not enter the input buffer. With respect to a TCP-based application, “urgent” messages can be seen as an additional buffer of size 1 in each direction that can be read asynchronously; hence, an out-of-band signal as demanded by the Socket API.

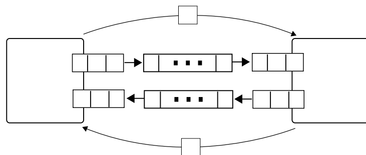


Fig. 1. Message Buffer Representation for TCP

3 Berkeley Sockets

As with different implementations of TCP, there are different “interpretations” of Sockets. To avoid confusion, we will restrict ourselves to Sockets for the Internet protocol suite and therein TCP. Further, we assume Sockets to work in *blocking mode*, i.e., that sending or receiving messages are atomic actions; this is the “standard” behavior available in all Sockets implementations. Note that Sockets' underlying programming paradigm makes a difference between clients and servers even if this seems strange for nowadays peer-to-peer applications.

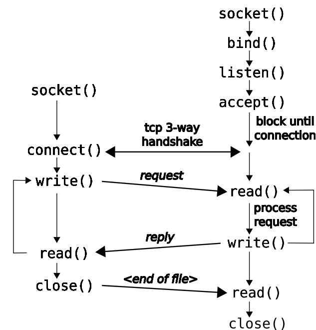
command	(natural language) semantics
`socket`	create a (local) socket of a type, e.g., a TCP socket
`bind`	bind local socket to IP-address and TCP port
`listen`	set underlying socket to listen mode (server)
`connect`	connect to remote socket (client, binding local socket implicit)
`accept`	accepts an incoming connection (server)
`read`, `write`	read, write from/to socket
`close`	remove socket

Table 1. Basic Sockets API Calls

Sockets includes different types of API calls for different classes of tasks: (i) establish and maintain a connection, (ii) send and receive messages, (iii) auxiliary methods to adapt to the local machine (e.g., converting endianness of IP addresses). Table 1 gives a natural language semantics of these calls; Figure 2

```
 1 #include <socket.h>
 2 /* overjump some lines ... */

 4 int main(int argc, char *argv){
 5     int sockfd, n;
 6     char recvline[MAXLINE+1];
 7     struct sockaddr_in servaddr; /* contains address data */

 9     if ((sockfd = socket(AF_INET, SOCK_STREAM), 0))<0)
10         err_sys("socket_error");
11     bzero(&servaddr, sizeof(servaddr));
12     servadr.sin_family = AF_INET;
13     servaddr.sin_port = htons(22);
14     servaddr.sin_addr.s_addr = inet_aton("12.110.110.204")

16     if (connect(sockfd, (SA*) &serveraddr, sizeof(servaddr))<0)
17         err_sys("connect_error")

19     /* we will only read ... */
20     while ((n=read(sockfd, recvline, MAXLINE)) >0){
21         recvline[n]=0;
22         if (fputs(recvline,stdout) == EOF)
23             err_sys("fputs_error");
24     }
25     if (n < 0)
26         error_sys("read_error");
27     exit(0);
28 }
```

Simple Server in C



Socket Functions of Elementary TCP Client-Server

Fig. 2. Socket Examples from [Ste04]

presents their practical usage in a simple server program and their interplay as sequence diagram.

4 Distributed Programs

Servers in the wild heavily rely on advanced features of Sockets and Tcp (like out-of-band data), multi-threading techniques that allow to fork a sub-thread (or even another process) that takes over the current Tcp-connection, pre-forking/-threading to avoid forking for every incoming connection, as well as asynchronous synchronization with the operating system (e.g., for reading/writing a file).

To simplify our presentation, we assume that peers are given by monolithic programs (single thread processes) with a formal (operational) semantics for all non-Sockets parts of the code. Further, we ignore the physical location of a program, as it is located by the sockets it uses. Since we regard Sockets for interprocess communication only, we can rule out self-loops, i.e., programs that directly communicate to itself. As before, we assume Socket calls to be blocking and leave aside out-of-band data. W.l.o.g., we assume the messages exchanged on connections to be over a finite alphabet. Note that the underlying Tcp assures that messages exchanged by Sockets Api calls are handled fifo and without loss or transportation error.

Let a *socket* be a pair of Ip-number and Tcp-port, i.e., two unsigned numbers of 32 and 16 bits each (for Internet Protocol version 4). A socket is *local* to a program if the program binds this socket (explicitly by `bind` or implicitly by `connect`).

Definition 4.1. *A* distributed program *is a tuple $\mathcal{DP} = \langle \mathbb{P}, sock \rangle$ with $\mathbb{P}$ a set of monolithic (local) programs. The injective function sock maps a program in $\mathbb{P}$ to a subset of pairs of sockets $(\varsigma_1, \varsigma_2)$ such that ς_1 is a local socket with respect to this program.*

Note that *sock* maps a program to all its outgoing/incoming connections which are given as socket pairs. This may require a full unfolding of the program. As connections are per default full-duplex there is no distinction between sockets that are used to either send or receive only.

Nevertheless, there are some artifacts introduced by the translation from (unformalized) Sockets to $\mathcal{DP}$. The mapping *sock* between processes and sockets is *static*, hence, cannot express the dynamic (re-)usage of sockets possible with TCP. Nevertheless, we can assume that our translation includes this aspects by bookkeeping different connections between the same sockets in time.

Further, we assume that each program in $\mathbb{P}$ has – a priori – an operational semantics with respect to all commands not in Sockets. We assert that this semantics is given as labeled transition system (LTS) wherein we treat Sockets actions as *nop*. This allows to use well-known techniques of model extraction, e.g., tools based on predicate abstraction or path-slicing like FeaVer [Hol00], abC [DHH02], Bandera [CDH+00], or JavaPathFinder[VHB+03].

5 Queueing Concurrent Processes

A *communication architecture* $\mathcal{T}$ (or architecture for short) is a pair $\langle \mathcal{P}, Ch \rangle$ with a finite non-empty set $\mathcal{P}$ of processes and a finite set of point-to-point channels $Ch \subseteq (\mathcal{P} \times \mathcal{P}) \setminus id_{\mathcal{P}}$.

Definition 5.1. *A system of* queueing concurrent processes (QCP) *over a given architecture $\mathcal{T} = \langle \mathcal{P}, Ch \rangle$ is a tuple $\mathcal{A} = \langle (S_p)_{p \in \mathcal{P}}, M, (\Sigma_p)_{p \in \mathcal{P}}, (\Delta_p)_{p \in \mathcal{P}}, (s_p^0)_{p \in \mathcal{P}} \rangle$ with M a finite message alphabet. For each process $p \in \mathcal{P}$, the tuple $\langle S_p, \Sigma_p, \Delta_p, s_p^0 \rangle$ describes a (local) transition system on the states S_p over the actions $\Sigma_p = \Sigma_p^{loc} \dot\cup \Sigma_p^{com}$ which are either local, i.e., in Σ_p^{loc}, or communication actions in $\Sigma_p^{com} = \{p!q(m) \mid (p,q) \in Ch \text{ and } m \in M\} \cup \{p?q(m) \mid (q,p) \in Ch \text{ and } m \in M\}$. Local transitions are given by the rules in $\Delta_p \subseteq S_p \times \Sigma_p \times S_p$, and the initial state of process p is s_p^0.*

As usual, $p!q(m)$ denotes the send of message m from process p to process q, whereas $q?p(m)$ denotes the matching receive on process q.

Note also that the S_p need not necessarily be finite. If all S_p are finite, we will call $\mathcal{A}$ a *finite* QCP. The local transition systems given by $\langle S_p, \Sigma_p, \Delta_p, s_p^0 \rangle$ could be, for example, finite automata, counter automata (including Petri nets), or pushdown automata. As usual, we define the semantics of $\mathcal{A}$ as labeled infinite-state transition system:

Definition 5.2. *A* QCP *$\mathcal{A}$ represents an* LTS *$[\![\mathcal{A}]\!] = \langle C, \Sigma, \rightarrow, c^0 \rangle$ with configurations $C = S \times (M^*)^{Ch}$ and the initial configuration $c^0 = (\mathbf{s}^0, (\varepsilon, \dots, \varepsilon))$, i.e., all channels are initially empty. We write a configuration as $c = \langle \mathbf{s}, \mathbf{w} \rangle$ where*

$\mathbf{s} = (s_p)_{p \in \mathcal{P}}$ is the global state and $\mathbf{w} = (w_{p,q})_{(p,q) \in Ch}$ are the channel contents. Further, for any $p \in \mathcal{P}$ and $a \in \Sigma$, $\langle \mathbf{s}, \mathbf{w} \rangle \xrightarrow{a} \langle \mathbf{s}', \mathbf{w}' \rangle$ is a transition in $C \times \Sigma \times C$ if $(s_p, a, s'_p) \in \Delta_p$ and the following holds:

(i) $s_q = s'_q$ *for all* $q \neq p$,
(ii) *if* $a \in \Sigma_p^{loc}$ *then* $\mathbf{w} = \mathbf{w}'$,
(iii) *if* $a \in \Sigma_p^{com}$ *with* $a = p!q(m)$ *then* $w'_{p,q} = w_{p,q}m$ *and* $w'_{s,t} = w_{s,t}$ *for* $(s,t) \in Ch \backslash \{(p,q)\}$,
(iv) *if* $a \in \Sigma_p^{com}$ *with* $a = p?q(m)$ *then* $w_{q,p} = mw'_{q,p}$ *and* $w'_{s,t} = w_{s,t}$ *for* $(s,t) \in Ch \backslash \{(q,p)\}$.

Note that the rules regarding Σ^{com} force a fifo semantics on the channels.

6 Formal Matching

We translate a given a distributed program $\mathcal{DP} = \langle \mathcal{P}, sock \rangle$ to a QCP $\mathcal{A} = \langle (S_p)_{p \in \mathcal{P}}, M, (\Sigma_p)_{p \in \mathcal{P}}, (\Delta_p)_{p \in \mathcal{P}}, (s_p^0)_{p \in \mathcal{P}} \rangle$ over an architecture $\mathcal{T} = \langle \mathcal{P}, Ch \rangle$ as follows:

- each program in $\mathbb{P}$ is mapped one-to-one to a process in $\mathcal{P}$
- $(p, p') \in Ch$ iff there exist two sockets ς, ς' such that $(\varsigma, \varsigma') \in sock(p)$ and $(\varsigma', \varsigma) \in sock(p')$.
- the local transition system $\langle S_p, \Sigma_p, \Delta_p, s_p^0 \rangle$ is an "adequate" semantic model for the program mapped to process p (when treating all actions in Σ^{com} as local *nop*)
- M is the finite alphabet of messages exchanged in $\mathcal{DP}$
- a write action of a message m on a connection (ς, ς') is mapped to $p!q(m)$ with $(\varsigma, \varsigma') \in sock(p)$ and $(\varsigma', \varsigma) \in sock(q)$
- analogously, the matching reception of message m is mapped to $q?p(m)$

Note that each connection in $\mathcal{DP}$ is mapped to a pair of channels with different directions in Ch. Hence, QCP models of $\mathcal{DP}$ are inevitably cyclic.

As discussed in Section 2, reliable and unbounded fifo channels seem a natural choice to model connections in distributed programs. Reliability is guaranteed by TCP and the assumption of unbounded channels avoids any occupation with the dynamic change of sliding window sizes.

Conclusions

The previous short report shows a possible way to use a fifo semantics for Berkeley Sockets based distributed applications. This further allows to use QCP as abstract model and verification methods based thereupon.

For future implementations, a more detailed semantic discussion of $\mathcal{DP}$ and its mapping to QCP is unavoidable. Further, one should try to reduce the implicit and explicit assumptions and restrictions included by our modeling and to extend the model to include the dynamic behavior of TCP as well as specialties like out-of-bound data.

References

[BFN+05] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *SIGCOMM*, pages 265–276. ACM, 2005.

[BSD83] 4.2BSD, 1983.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, pages 439–448, 2000.

[DHH02] Dennis Dams, William Hesse, and Gerard J. Holzmann. Abstracting c with abc. In *Proc. of CAV 2002*, pages 515–520, 2002. Springer.

[CMGMS05] Pedro de la Cámara, María del Mar Gallardo, Pedro Merino, and David Sanan. Model Checking Software with Well-Defined APIs: the Socket Case. In *Proc. of FMICS 2005*, pages 17–26, 2005. ACM.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol00] G. J. Holzmann. Automating software feature verification. *Bell Labs Technical Journal*, 5(1):35–45, March 2000.

[HS02] G.J. Holzmann and M.H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.

[RFC793] Transmission Control Protocol. RFC 793 (Informational), 1981.

[RFC1323] TCP Extensions for High Performance. RFC 1323 (Informational), 1992.

[Ste95] W. Richard Stevens. *TCP/IP Illustrated, Volume 1; The Protocols.* Addison Wesley, Reading, 1995.

[Ste04] W. Richard Stevens. *UNIX Network Programming Vol 1: Networking APIs – The Sockets Networking API*, volume 1 of 3. Prentice Hall, third edition, 2004.

[VHB+03] W. Visser, K. Havelund, B. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.

[WNSS02] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing udp: Mechanized semantics for sockets, threads, and failures. In Le Métayer, editor, *Proc. of ESOP 2002*, pages 278–294. Springer, 2002.