2nd VerifyThis Long-term Challenge: Specifying and Verifying a Real-life Remote Key-Value Cache (`memcached`)

Gidon Ernst[1] and Alexander Weigl[2]

[1] Ludwig Maximilian University, Munich, gidon.ernst@lmu.de
[2] Karlsruhe Institute of Technology, weigl@kit.edu

Abstract. Tremendous progress was achieved in the area of deductive program verification in last decades. In the Verify This Long-term Challenge we want to demonstrate our performance in achieving safe and secure software systems. Goal of the challenge is to foster collaboration in order to verify a realistic and industrially-relevant software application. This paper introduces the second Long-term challenge: The specification and verification of a remote key-value cache, inspired by and acting as compatible drop-in replacement of the `memcached` software package, which is widely in use at major companies.

Website: https://verifythis.github.io/
Mailing List: https://www.lists.kit.edu/sympa/info/verifythis-ltc
Reference System: http://memcached.org/

1 Introduction

Program verification is an established and important research area with tremendous progress since its beginning in Weakest-Precondition- and Hoare calculus [8]. Witnesses of development of can be found in large large, non-trivial case studies, such as the verification of the data structure and algorithm implementation in the shipped Java libraries [5,4], or the parallel nested depth first search [16]. State-of-the-art program verification tools compared and evaluated in competitions such as VerifyThis [12,13,11,9,10] and VSComp [15,7], which both embrace the "human factor" [6] as a necessary ingredient to specify and verify complex, application-specific properties of software. This is in contrast to SV-COMP [3] and Test-Comp [2], for example, which aim at evaluating fully automatic methods (such as explicit and symbolic model-checking) for software verification and testing instead. Nevertheless, the challenge proposed in this document contains many aspects that can be tackled with tools typically participating at SV-COMP and Test-Comp and we explicitly welcome contributions in this regard.

The intention of the VerifyThis Collaborative Long-term Challenge series is to demonstrate that deductive program verification can produce relevant results for real systems with acceptable effort. Moreover, we aim to bring together members of the community for a focused exchange of ideas, techniques, insights, and

experiences. The developments will serve to evaluate and improve the effectiveness and maturity of verification tools.

Here, we propose the **second VerifyThis long-term challenge**: A verified drop-in replacement of the in-memory key-value cache `memcached`.[3] Software like `memcached` is the backbone for fast response in cloud-native environment by storing and caching *hot* information. This particular software package is open source (BSD 3 clause license) and it is widely in use at major players like Google, Amazon, Microsoft, Facebook, and Twitter.

At the same time, much of the complexity of `memcached` is internal, i.e., its external interface is fairly straight-forward, which means that developing a (verified) drop-in replacement that supports a compatible subset of the protocol requires a reasonably low effort only.

The intuitive understanding of `memcached`, described in detail in section 2, is that of a mapping from keys to values. However, `memcached` is *not* adequately described by a simple functional mapping, i.e., it is not "just another hash-table challenge". It comes with some characteristics that render it interesting from a specification perspective, notably centered around the semantics of protocol actions and cache eviction. This is paired with elaborate internal, implementation-level data structures and algorithms, which bring, e.g., concurrency and memory management into the picture. There is a wealth of (typical) conceptual challenges, such as integration with unverified OS interfaces and how to achieve a modular architecture in which abstractions are not leaky.

The overarching goals and research directions of this long-term challenge, further elaborated in section 3, are as follows:

- Develop high-level behavioral models and contract specifications that abstractly capture the core functionality of a remote cache server, the protocol, and the client library
- Characterize global properties of the entire system, e.g. temporal liveness and safety properties related to the lifecycle of cache entries
- Design and verify an implementation that realizes these requirements and that may serve as a drop-in replacement for `memcached` with support of a significant subset of its features
- Verify parts of the actual `memcached` implementation (written in C), using e.g. scalable software model checking methods or focused deductive techniques on critical routines

We emphasize that the challenges associated with these respective goals can be scaled in many dimensions (realistic interfaces, algorithms and data structures, features). It is therefore *easy to get started*, in fact, we provide two abstract but reasonably complete executable reference implementations that should help with the first steps (section 2).

At the same time there is a clear perspective for a scientifically and practically meaningful outcome. We welcome collaboration and contributions of any kind—if you are interested, please join the Mailing list and let us know of your intention

[3] https://memcached.org/

to participate. We will follow-up with informal community meetings co-located to major events and we are planning a joint publication effort to document the outcomes (cf. section 4).

2 System Description

`memcached` is an in-memory key-value store that acts as a cache. It offers operations to enter keys into the cache data structure with associated values and also a timeout for which the association is valid. In contrast to traditional databases, entries can implicitly be evicted from the cache due to memory pressure, which `memcached` resolves by a LRU protocol. The architecture of the main `memcached` application is that of a server which serves requests to clients by spawning multiple threads, which in turn access the shared internal data structure. The standard interface is a simple text and line-based over telnet with a small set of commands that include lookup, update, and also (atomic) replacement. `memcached` is supposed to serve multiple front-end applications at a time. Typically, `memcached` is used via a client library interface which offers a high-level API. Such libraries exist for a variety of programming languages (including Javascript and PHP). Besides setting up the telnet connection and realizing the simple communication protocol, client libraries may also support load-balancing queries over a pool of `memcached` servers, such that each server is made responsible for a subset of the keyspace in use by the application.

Resources: Please feel free to use the following resources to familiarize yourself with the functionality and internals of the `memcached` server and its behavior. Note that both implementations may be incomplete and/or not entirely be faithful to the reference `memcached`. If you spot such a difference that is not documented please contact the respective author or file a github issue.

- A great description of the internals is this video by Hussein Nasser: `memcached` is `https://www.youtube.com/watch?v=NCePGsRZFus`
- High-level executable Python model and implementation (Gidon Ernst): `https://github.com/gernst/pycached`
- Java implementation (protocol, main functionality, Alexander Weigl): `https://github.com/wadoon/bloatcache`

Figure 1 shows a high-level diagram of the architecture of the `memcached` server and how it communicates with clients via programming libraries. We describe the text protocol in section 2.1, some aspects of the internal storage system in section 2.2, and some aspects of client libraries in section 2.3. We defer specification and verification challenges to section 3.

2.1 Text Protocol

The communication between the clients and server takes place over TCP/IP protocol in which both parties are sending message. A message consists of either

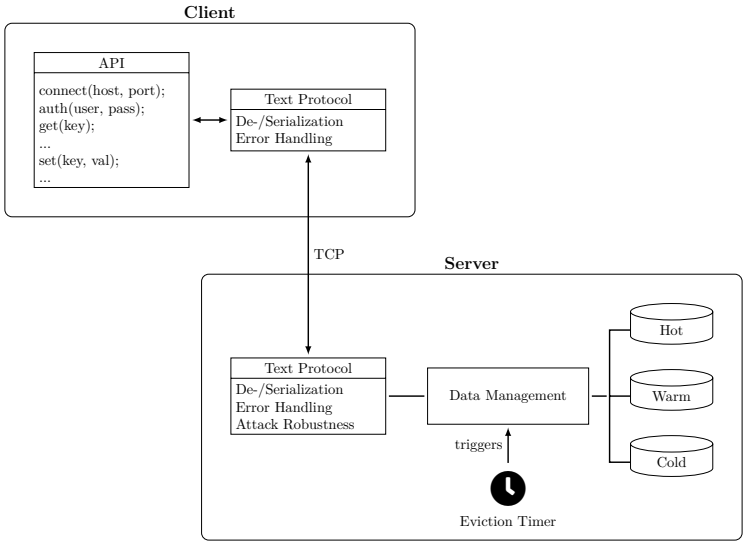

Fig. 1. Architecture diagram: The client uses an abstraction which provides access to server functionalities. Requests and responses are exchanged using a text protocol. The server manages the parallel access on the three buckets of the LRU cache.

one or two lines, which are terminated by carriage return `\r` and newline feed `\n`. The communication is a mix of text in ASCII encoding (commands, keys, meta-data) and binary (values, also "content" below). Each command line is similar to a shell input. The command line consists of command name (lower case and case-sensitive) and the parameters delimited by a space (ASCII char 32). Messages that specify a value explicitly mention its length in bytes.

Note that here we concentrate on the functional necessary commands. Helper or debugs commands (e.g., statistic query) are omitted to give us more flexibility on the data structure. The official documentation of the protocol can be found here: `https://github.com/memcached/memcached/blob/master/doc/protocol.txt`

Examples of communication are in appendix A. The majority of this section focuses on the protocol, because it represents the external interface as a common ground for all efforts.

Requests. The client is able to send following retrieval or manipulation commands. The connection is can always be closed by the client without sending a termination command. `memcached` knows four retrieval commands and also eight manipulation commands. The syntax is given in Fig. 2 and 3:

retrieval The `get` and `gets` request the stored value of the given keys. The answer for a key is defined in Fig. 5. Additionally to `get`, `gat(s)` also updates the expiration time of a stored key-value pair.

`touch` The update of the expiration time can be done by `touch` without requesting the current value.

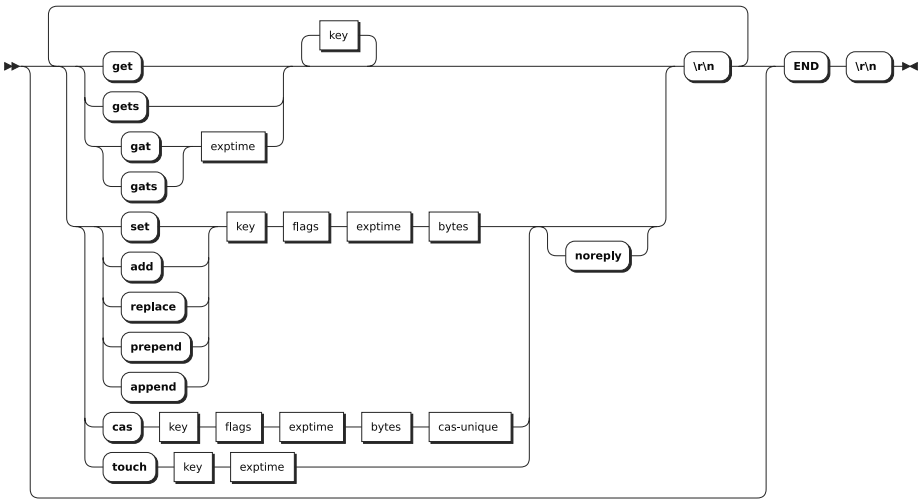

Fig. 2. Syntax diagram for client requests

```
req  ::= (<ret> | <man>)*
end  ::= "END" "\r\n"
ret  ::=  (<gat> | <get>) "\r\n"
get  ::= ( "get" | "gets" ) <exptime> <key>*\r\n
gat  ::= ( "gat" | "gats" ) <exptime> <key>*\r\n

man  ::= <cas> | <touch> | <store> | <delete>
store ::= ("set" | "add" | "replace" | "prepend" | "append")
         <key> <flags> <exptime> <bytes> ["noreply"] "\r\n"
cas  ::= "cas" <key> <flags> <exptime> <bytes> <cas unique>
         ["noreply"] "\r\n"
touch ::= "touch" <key> <exptime> ["noreply"]"\r\n"
delete ::= "delete" <key> ["noreply"]\r\n
```

Fig. 3. Grammar in EBNF of client requests

manipulation The five manipulation (`set`, `add`, `replace`, `prepend`, `append`) commands have the same structure. The client supply the key, possible flags, expiration time and the length of the content, followed by the content on the next line. The behavior of the handling depends then on the command. `set` just assigns the given content, expiration time and flags to the key regardless of the current state. `add` does the same, but only of if the key is unassigned. `replace` only reassign existing keys. `append` and `prepend` append or prepend the given content on the previously assigned value.

`incr`/`decr` The increment and decrement are a little bit simpler: the given value is added or subtracted to the assigned value to the key. For this, the assigned value is treated as a 64-bit unsigned integer. If the assigned value does not fit into this scheme, an error is thrown.

There is no underflow: If decrement reduces the value below zero, it becomes zero. Overflows are still possible.

`cas` The command `cas` stands for compare and swap, which bases on the unique *cas* value. `cas` is the base for parallel processes. `cas` replaces (swaps) a value only if the supplied cas number is the same as the internal stored one. This helps to prevent race conditions between multiple client. The internal cas number is maintained on changes of the key-value entry.

`delete` The command `delete` removes an item instantly from the internal cache.

Any *number*, such as a timestamp or a length is represented by its human-readable ASCII decimal encoding.A *key* is a sequence of bytes with a length at most 250 bytes, and without control or whitespace characters (as regular expression `/[!-ÿ]{1,250}/`). The *exptime* defines the expiration time of an entry in seconds. Given values smaller or equal to 2592000 (seconds in 30 days), the given expiration time is considered relatively to the current time, otherwise it is seen as the absolute UNIX timestamp (seconds since 00:00:00 UTC on 1 January 1970). The *flags* are the decimal encoding of an unsigned 16-bit number (32-bit in newer versions) that is opaque to the server and can be used by the client. The *bytes* are the decimal encoding of an unsigned integer describing the length of the values. There may be an implementation-defined upper bound on the length of values, and it seems that `memcached` supports content of up to 1 megabyte (cf. SLAB classes in section 2.2). A reasonable upper limit is that the length can be represented as a 64-bit unsigned integer. The *cas-unique* is the decimal representation of an 64-bit unsigned integer which identifies the serial number of a value that is associated to (any) key. `memcached` appears to have a global counter that increments with each mutation command and is stored as part of an entry as its "serial number". It is thereby possible to detect whether the value for a given key is unchanged by storing the *cas-unique* and comparing it on later accesses.

Usually, each command is acknowledged explicitly by the server (see below). The `noreply` keyword can be attached to some of the commands to signal that no response should be emitted from the server.

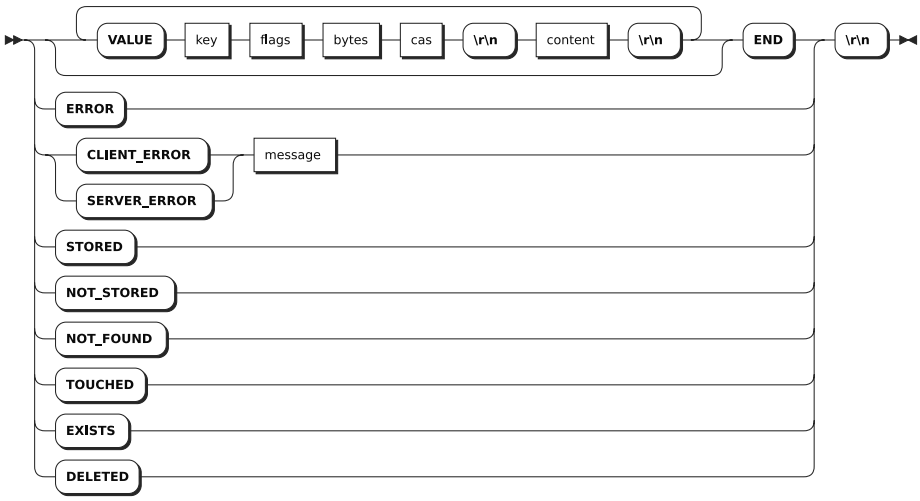

Fig. 4. Syntax diagram for responses

```
answer     ::= (<vals> | <error> | <success> | <value>)
vals       ::= <val>* <end>
val        ::= "VALUE" <key> <flags> <bytes> [<cas-unique>]\r\n
               <content>\r\n

error      ::=  "ERROR\r\n"
              |  "CLIENT_ERROR [<error>]\r\n"
              |  "SERVER_ERROR <error>\r\n"

success    ::=  "STORED\r\n"
              |  "TOUCHED\r\n"
              |  "NOT_FOUND\r\n"
              |  "NOT_STORED\r\n"
              |  "EXISTS\r\n"
              |  "DELETED\r\n"
```

Fig. 5. Grammar in EBNF for server-side responses

Responses Given a request of a client, the server answers retrieval commands with a list of value entries, error or success messages, and in case of `incr` and `decr` with the new value. The overview is given in figs. 4 and 5.

The retrieval commands `get`, `gats`, ... return a list of value entries. A value entry consists of two lines: The first is indicated by the keyword `VALUE` and followed by key, current flags, the length of the content and the internal cas value. Followed by the content of the previously transmitted length in the first line. Requested key entries that are now found are silently omitted.

The manipulation commands terminate successful with the message `STORED`. In case of `touch` the answer is `TOUCHED`. And for `incr` and `decr` the new value is returned. `NOT_FOUND` signals that the entry was expected to exists, but could not be found (e.g., required by `incr` or `replace`). `NOT_STORED` signals that the entry was not updated (e.g., `add` command on an existing entry).

Additionally, the server can answer with an error, signaling that something is unexpected happened, e.g., a wrongly formatted command. The general error is just the message `ERROR`. More detailed are delivered with `CLIENT_ERROR <error>` which signals that the client is responsible for the error. `SERVER_ERROR <error>` indicates a server-side error. After an `SERVERE_ERROR` the server closes the connection. This is the only case, in which the server terminates a connection with a client.

Authentication `memcached` supports username-password authentication, which is implemented by using the `set` command. If authentication is activated the first command must be `set` with the content of the username and password:

```
set <key> <flags> <exptime> <bytes>\r\n
<username> <password>\r\n
```

The parameters `key`, `flags` and `exptime` are ignored, but `bytes` has to match the content length. The server answered regularly with a `STORED` or `CLIENT_ERROR`.

2.2 `memcached`'s Storage Oganization

This section describes how `memcached` organizes its storage internally. Solutions to the long-term challenge, however, may deviate from the scheme described below in any regard. The description is necessarily very low-level, to reflect the design decisions made for space and time efficiency.

Entries are maintained in a cache data structure (the SLABs, see below) indexed by a hash table from keys pointing to entries in the cache. In `memcached`, the hash-table is of dynamic size with linked lists to collect entries with the same hash.

Entries stored by `memcached` are partitioned according to the size of the attached values. There are bins, so-called SLAB classes, for exponentially increasing sizes, such that for example, all entries with values of length n with $2^k \leq n < 2^{k+1}$ are stored in the k-th SLAB class. A SLAB then consists of a number of operating system pages, i.e., it is possible to resize a SLAB with page granularity.

Entries are removed from the cache data structure in two situations: Explicitly after their expiration time (by a background thread that scans the cache) and implicitly when memory pressure is encountered. In the latter case, `memcached` evicts those entries which are not used often. `memcached` implements two schemes for that purpose.

In the old scheme, the entries in each SLAB form a doubly linked list in *last-recently used* LRU order—each access moves that entry to the beginning of the LRU list. Therefore, entries at the end of the LRU list are good candiates for cleanup, because they seem to be used less frequently.

In the new scheme, the cache is partitioned into HOT, WARM, and COLD areas, in which unused entries gradually move to cooler areas. Space reclamation is performed on the COLD area (first?). The details can be found in this document: https://github.com/memcached/memcached/blob/master/doc/new_lru.txt

2.3 Client Libraries

A client library is an API in a particular programming language that offers a high-level and perhaps type-safe way to access `memcached`'s functionality. This may be realized by an interface or similar with top-level methods/procedures for all commands listed in section 2.1 over appropriate data types that represent keys, values, timestamps, and so on.

Such a library is a place where a multiplexing proxy can be implemented, i.e., a mechanism that partitions a (huge) number of entries across multiple servers, each running `memcached`. The client would then simply partition the key-space according to the number of servers and select the appropriate one to access a given key.

3 Specification and Verification Challenges

Goal: High-level Behavioral Models

The first challenge is to create a faithful formal specification of the external behavior of `memcached` at a suitable degree of abstraction. Part of such a specification is a set of operations, an abstract representation of the *externally data model* (such as keys, values, ...) and an abstract representation of the *internal state*. The Python and Java implementations mentioned as preliminary work in this document are already fairly close to such a model, however, they are not fully formal.

Clearly, there are many ways to approach this challenge. For example, one has the choice of whether the model is state-finite, whether nondeterminism is used as a specification mechanisms, whether the model is executable, and how accurately cache timeouts are reflected. This kind of specification serves different purposes in the challenge:

Documentation of the functional requirements, as an anchor point and reference for all further development. Note that it can be shared across the

specification of the server operations and the client API, for example, with the actual protocol implementation being a separate concern.

Modularization: Typically, in formal developments, one must pay close attention that the structural abstractions support the subsequent verification. By emphasizing the role of the behavioral model as a separate entity here, we also urge to keep the top-level specification clean of any implementation concerns, such exposing internal pointers or mixing implementation-level objects into the description of inputs and outputs of operations.

Collaboration can be achieved if such a specification is formulated with a mathematical language that can be translated without much ambiguity between tools. A common ground is SMT-LIB like, i.e., first order logic with sets, maps, sequences, and records, possibly with function definitions and quantifiers, wherein the semantics of system operations is captured as relations between inputs, outputs, and a pre-/post pair of internal abstract states. While we do not necessarily suggest that this kind of collaboration is made manifest in an *automatic* translation between tools and formalisms, we hope it will take a step towards unifying specification languages and integrating verification tools eventually.

Goal: Characterize Global Properties

Another step to characterize the intended behavior of the system is to focus on global properties that go beyond simple two-state specifications or contracts, e.g., trace-properties expressed in LTL or CTL, or even hyperproperties over multiple traces. Closely related is the question of how to adequately model the passing of time. Again, this kind of specification has different aspects:

Documentation of desired behavior by relating different events and operation calls. For instance, a property of interest could be that whenever a cache entry is stored with an expiration date, it *can* be retrieved successfully before the timeout (an example for a liveness property) and it becomes inaccessible afterwards (a safety property)—unless of course the entry for the same key is overwritten or it is evicted due to memory pressure.

Collaboration between deductive methods and model-checking approaches hinges on how a contract-based view can be connected to such a high-level perspective. Research questions include for example how to map between events and the code, how to integrate the respective semantic foundations, and under which conditions the properties that have been verified are preserved at the implementation level (e.g. via refinement).

Goal: Design and Verify an Implementation

Implementing a cache server that adheres to the behavioral model may embrace as little and as many of the strategies of `memcached` for efficiency, such as concurrency or explicit memory allocation and management. Some interesting aspects are:

A verified **Core System** which implements the top-level specification of desired behavior.

Protocol: A key challenge is a verified implementation of the parser/serializer that bridges between a conceptual representation of protocol messages and the actual text/byte-based representation that is communicated between the server and the client. For example, is it possible to provide a systematic and declarative definition of the on-wire format via a grammar and can the verification make effective use of such a specification?

Integration with unverified operating system interfaces. Specifying the behavior of network interfaces (like sockets) and the respective functionality offered by the OS and programming environment comes with the need of having some abstract model of the underlying network connection, i.e., the other end of the communication that is accessed by the server. While these concerns may typically be fairly low-level, one can also ask: what are the assumptions that are needed to prove the high-level properties, specifically which assumptions on the communication partner are needed to make liveness properties true, and how can such assumptions be represented? Moreover, different programming paradigms exist for multiplexing serving multiple clients at a time, such as threads, fork-and-exec, callbacks, or non-blocking input and output, which may involve non-sequential control flow.

Goal: Verifying the `memcached` implementation

Lastly, we propose to look at verifying (parts of) the actual open source implementation `memcached`. Such efforts can roughly be distinguished into two categories: **Software Model Checking** of generic properties like memory safety or absence of race conditions of the actual C source code. **Deductive Verification** of critical components, data structures, or algorithms. In both cases, it is of interest to know how much effort is needed to adapt the original source code for the purpose of verification, or viewed in reverse, how well existing tools can deal with real-world code and the C features used in `memcached`. A proposal is to derive suitable benchmark tasks for the SV-COMP benchmark repository [3].[4]

List of Technical Challenges

Below we summarize some technical challenges that we think are interesting. We do not claim that this list is exhaustive and we welcome feedback by participants to extend this list with additional properties, aspects, and requirements that we have not yet thought of.

Server Side Goal is to develop a drop-in replacement of the `memcached` server that is compatible at the interface level. There are many variations and challenges which can be addressed (e.g. for efficiency), but a simple server can be realized rather quickly to get started.

- Implementation and verification of the text-based protocol, including the parser and serializer, and mapping to some higher-level data types

[4] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

- Implementation and verification of the network management code, including set-up of the listener socket and serving of incoming of connections
- Integration of the program with unverified OS interfaces
- Implementation and verification of an in memory hash table that backs up the key value store.
- Implementation and verification of the memory allocator for the stored data, which in `memcached` is realized as a SLAB to deal with fragmentation issues
- Realization of the cache eviction protocol and timestamps. In `memcached`, the caches are partitioned into HOT, WARM, and COLD, with the expecation that the HOT items are read a lot, whereas the COLD area is scanned for stale/out of date data
- Implementation and verification of concurrency inside the server, perhaps making use of some fancy lock-free techniques
- DDOS protection
- Model and verify performance in terms of $O(\_)$ for space/runtime efficiency for all data structures/algiorithms

Client Library The goal is to develop a client library for your favorite programming language that provides higher- level abstractions over the lower-level network protocol.

- Protocol and connections (cf. above)
- Develop a high-level specification of the API which can be relied on by application programs
- Support to act as a proxy to multiple back-end `memcached` servers by partitioning the keyspace
- Perhaps: load-balancing features
- Memory safety and all that
- Functional Correctness of the various aspects
- Runtime and memory complexity/bounds (e.g. cache size, hash table amortized costs)
- Temporal properties, such as that each connection is served (under some fairness assumptions)
- OS interface specifications, modeling of the network communication
- Global LRU eviction protocol and cache timing guarantees (e.g. no entry is stored longer than the user-supplied validity timeout)

4 Participation and Time Schedule

The first long-term challenge did take place during the COVID-19 pandemic, which gave us the advantage of establishment of online meetings, which were held frequently. For the 2nd LTC we propose following schedule: The announcement of the challenge takes place at the TOOLympics session at the ETAPS conference. A first informal meeting is intended to take place at the iFM in Leiden in 2023 and a second meeting at the ETAPS in April 2024. We intend to close the challenge with a track at ISoLA in autumn 2024.

To participate, please register on the VerifyThis LTC Mailing List: https://www.lists.kit.edu/sympa/info/verifythis-ltc, over which all communication and discussion takes place. To make your team known, please send a short E-Mail to announce your participation, perhaps including a plan on which features you want to work on, which programming language you target, which verification tool(s) and approaches you plan to use, and whether you are seeking collaboration on specific aspects. The organizers will also add your team to the list of participants on the website.

References

1. Wolfgang Ahrendt, Paula Herber, Marieke Huisman, and Mattias Ulbrich. Specifythis - bridging gaps between program specification paradigms. In *ISoLA (1)*, volume 13701 of *Lecture Notes in Computer Science*, pages 3–6. Springer, 2022.
2. Dirk Beyer. Advances in automatic software testing: Test-comp 2022. In *FASE*, volume 13241 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2022.
3. Dirk Beyer. Progress on software verification: SV-COMP 2022. In *TACAS (2)*, volume 13244 of *Lecture Notes in Computer Science*, pages 375–402. Springer, 2022.
4. Martin de Boer, Stijn de Gouw, Jonas Klamroth, Christian Jung, Mattias Ulbrich, and Alexander Weigl. Formal specification and verification of jdk's identity hash map implementation. In Maurice H. ter Beek and Rosemary Monahan, editors, *Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings*, volume 13274 of *Lecture Notes in Computer Science*, pages 45–62. Springer, 2022.
5. Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying openjdk's sort method for generic collections. *J. Autom. Reasoning*, 62(1):93–126, 2019.
6. Gidon Ernst, Marieke Huisman, Wojciech Mostowski, and Mattias Ulbrich. Verifythis - verification competition with a human factor. In *TACAS (3)*, volume 11429 of *Lecture Notes in Computer Science*, pages 176–195. Springer, 2019.
7. Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump. The 2nd verified software competition: Experience report. In Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*, volume 873 of *CEUR Workshop Proceedings*, pages 36–49. CEUR-WS.org, 2012.
8. Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 345–373. Springer, 2019.
9. M. Huisman, R. Monahan, W. Mostowski, P. Müller, and M. Ulbrich. VerifyThis 2017: A program verification competition. Technical report, Karlsruhe Reports in Informatics, 2017.
10. M. Huisman, R. Monahan, P. Müller, A. Paskevich, and G. Ernst. VerifyThis 2018: A program verification competition. Technical report, Inria, 2019.
11. M. Huisman, R. Monahan, P. Müller, and E Poll. VerifyThis 2016: A program verification competition. Technical Report TR-CTIT-16-07, Centre for Telematics and Information Technology, University of Twente, Enschede, 2016.

12. Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis 2012. *Int. J. Softw. Tools Technol. Transf.*, 17(6):647–657, November 2015.
13. Marieke Huisman, Vladimir Klebanov, Rosemary Monahan, and Michael Tautschnig. VerifyThis 2015. A program verification competition. *Int. J. Softw. Tools Technol. Transf.*, 19(6):763–771, 2017.
14. Marieke Huisman, Raúl Monti, Mattias Ulbrich, and Alexander Weigl. The verifythis collaborative long term challenge. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, chapter 10, pages 246–260. Springer, December 2020.
15. Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
16. Wytse Oortwijn, Marieke Huisman, Sebastiaan Joosten, and Jaco van de Pol. Automated verification of parallel nested DFS, 2019. Submitted.

A Communication Examples

The following listing is an example of the communication between a client and the `memcached` server. Lines starting with "`>`" are sent by the client, lines marked with "`<`" are sent by the server. We added line comment marked by `#` given further information. `\r\n` are represented by line breaks.

```
# telnet localhost 11211

> set abc 16 120 5   # flag: 16, TTL: 120 s, len: 5 bytes
> 12345
< STORED

> gets abc def        # request two keys
< VALUE abc 16 5 18   # flag: 16, len: 5, cas: 18
< 12345
< END                 # Note the missing response for key "def"

> incr abc 10
< 12355
> decr abc 10
< 12345

> append abc 8 120 1
> 0
< STORED
> get abc
< VALUE abc 16 6     # get does not return cas value!
< 123450
< END

> prepend abc 4 120 1
> 9
< STORED
> get abc
< VALUE abc 16 7
< 9123450
< END
```

B History of the VerifyThis and the Long-term Challenges

VerifyThis on-site events comprise short-term exercises with strict time constraint, typically 1–2h, with a focus on competing. In contrast, with this long-term challenge, we want to foster collaboration between the participants to show what can finally be done by using program verification techniques in a mission-critical realistic, industrial-size software.

The first VerifyThis long-term challenge [14] was dedicated to HAGRID, a verifying key server for public PGP keys; it is the successor of the SKS key server, which has multiple security vulnerabilities, for example uploading PGP keys for arbitrary user identities. HAGRID introduces a side-channel (E-Mail) to verify the ownership of an email address to address this issue, such that the published association between an E-Mail address ("identity") and keys on the server agrees with the intention of the account holder. The outcome of this challenge is a number of working prototypes, system specifications and models, as well as proofs, which together illustrate the various properties like functional correctness and security.

The Casino challenge[5] arose from a series of online discussions. It is about security of a block-chain smart contract that models a simple coin guessing game, in which players can bet money against the operator of the game. The challenge has sparked many interesting contributions, which looked into aspects specific to block-chain, such as the way state is stored permanently and publicly and how the transaction-based execution semantics and re-entrant interact with respect to the transfer and potential loss of money. An overview can be found in [1].

There are two take-aways from the past challenges, which will serve as key guiding factors in the choice and setup of the second long-term challenge:

- A wide range of approaches and tools can contribute very different and complementary insights into a specification and verification problem like the Casino.
- As a consequence, system-level models and specifications (e.g. in terms of automata or traces) are contrasted to state-based methods and contract-based approaches (in terms of pre-/post relations)

To accommodate both, we want to foster the development of approaches resp. tools that can deal with both views in a more integrated way. More importantly, we want to encourage that the *collaboration* between participating teams is structured around these two aspects as outlined in this document.

[5] https://verifythis.github.io/casino/