Native SSH and SFTP Support in Unicon

Jafar Al-Gharaibeh

Unicon Technical Report: 26

August 2026

Abstract

Unicon historically had no native SSH client: programs that needed remote command execution or file transfer shelled out to an external ssh binary. This report describes the current design and implementation: a client that binds libssh and exposes sessions, multiplexed channels, and SFTP through existing language verbs – open(), read()/write(), receive(), subscript peek, stat(), remove(), and rename() – rather than a parallel SSH-specific API. The report covers the connection and channel model, authentication and attributes, examples and use cases, the three I/O tiers, interactive shells, SFTP, build integration, and remaining work.

Keywords: Unicon, SSH, SFTP, libssh, sockets, runtime, technical report.

Unicon Project
http://unicon.org
University of Idaho
Department of Computer Science
Moscow, ID, 83844, USA

1 1. Introduction

This report describes the current design and implementation of native SSH in the Unicon runtime: how a session is opened, how additional channels and SFTP handles are derived from it, how stream I/O, ordered events, and metadata operations are exposed, and what remains out of scope. It is formatted as a Unicon Technical Report [1]. The language-facing reference lives in *Programming with Unicon* (Chapter 6, "Secure Shell") and the language reference (`open` mode `h`, subscript peek, `key()`, `receive`, `stat` / `remove` / `rename`). section 6 collects examples by mode and use case. Automated tests live in `tests/posix/ssh.icn`.

The feature is optional. A build with libssh present reports `secure shell` in `&features` (the `_SSH` feature flag). Without the library, mode `h` is ignored and the SSH paths are compiled out, as with other optional features.

2 2. Motivation and prior art

Unicon programs that needed SSH previously relied on `popen()` or pipes to an external `ssh` process. That is not portable (Windows often has no OpenSSH client in `PATH`) and gives the program no structured access to channels, exit status, or SFTP. Native primitives let an application authenticate to hosts, run commands, and transfer files without an external process.

2.1 2.1 Library choice

Two C libraries were considered. **libssh** (LGPL) supports both client and server, modern key exchange and host keys, and true non-blocking I/O via `poll(2)`. **libssh2** (BSD) is client-only and cannot be used fully non-blocking. The implementation binds **libssh** [2], for server-side optionality later and for the callback API that preserves stdout/stderr arrival order (section 8.3).

A full protocol reimplementation (Paramiko, Go's `x/crypto/ssh`) would mean owning crypto primitives and protocol-drift / CVE tracking indefinitely. Binding to libssh via the existing C runtime avoids that.

2.2 2.2 Open as the verb

Python (Paramiko) and Go (`x/crypto/ssh`) both treat "new session" as sugar over "open another channel on this connection." The verb is **open**, applied to an existing connection object, not "clone" or "duplicate." Unicon already has `open()` as the universal constructor for files, sockets, SSL, and messaging connections. `Clone()` is the Graphics-facility naming convention; a distinct `channel()` function would add a new global name for an operation `open()` already expresses.

2.3 2.3 Why not Messaging

The Messaging facility (HTTP/POP/SMTP, `rmsg.r`, libtp) [3] is not a fit. Messaging's transport-discipline abstraction has the same 1:1 single-stream limitation as SSL – no channel-multiplexing concept – so moving into that world would not solve the multiplexed-channel requirement, only relocate it. SSH's request/channel model (`exec` / `shell` / `sftp` as concurrent channels) also does not map onto Messaging's request/response verb model (`GET` / `POST` / `RETR`). Instead, SSH uses the `h` mode character and plain `user@host` string form, independent of the Messaging/URI subsystem.

3 3. Architectural grounding

The implementation follows established SSL and socket patterns rather than inventing a parallel subsystem.

`union f` in `src/h/rstructs.h` is the file block's descriptor slot: a tagged union of `FILE *`, socket `fd`, `SSL *` when `HAVE_LIBSSL`, and so on, discriminated by status bits at runtime. SSH does **not** embed libssh objects directly in the union. It adds a pointer, `struct SSHfile *sshf`, matching the Messaging `MFile *` precedent. `SSHfile` is allocated with `malloc` so pointers to it are stable across garbage collection, and is freed by the close hook. A malloc'd block is required once the session must hold a list of child channels whose addresses survive compaction (section 10.2).

Status bits live in `src/h/rmacros.h`. `Fs_Encrypt` is `0200000000`; SSH takes the next bit, `Fs_SSH = 0400000000`. `open()`'s mode string is scanned character-by-character in `src/runtime/fsys.r`. `e` sets `Fs_Encrypt`. SSH uses `h` / `H` (`s` is already the Messaging short-request flag).

Attributes are `"key=value"` strings passed as trailing arguments to `open()`, parsed in `create_ssh_session()` the same way `create_ssl_context()` parses SSL attributes – not a fixed argument list. A leading integer, when present, is a connect timeout in milliseconds, as for plain sockets.

All stream I/O funnels through one dispatch point: `u_read()` / `u_write()` in `src/runtime/rposix.r`, which already branch on `Fs_Socket` / `Fs_Encrypt`. SSH adds a third branch for `Fs_SSH`, reading from the arrival-order queue (section 8) or `sftp_read()`, and writing via `ssh_channel_write()` or `sftp_write()`. Nothing above this dispatch (`reads()`, `read()`, and so on) needs its own SSH case, except where a line-oriented helper must not wait for a newline that will never arrive (section 9.2).

The close hook in `fsys.r` frees SSL resources directly. SSH files follow the same convention – leaking a channel or session means not calling `close()`, the same as any other file.

`receive()` already returns a general-purpose `posix_message` record (`addr`, `msg`, `saddr`, `daddr`, `ttl`, `proto`) for UDP and raw datagrams. SSH reuses that type for an ordered stdout/stderr/exit-status event stream, using `addr` and `msg` (section 8.3).

`stat(f)` already `type_case`s on its argument (string path versus open file). SFTP extends both branches (section 11.3). `remove()` and `rename()` gain an optional leading session argument, matching the existing Unicon idiom that many functions take an optional leading file or window (`write("hello")` versus `write(f, "hello")`).

4 4. Connection and channel model

4.1 4.1 `open()` dispatch

Two shapes, both using the existing `open()`. The second argument is the usual mode string; `h` is a new character in that scan, and `c` / `s` after `h` select a channel or SFTP the same way `u` after `n` selects UDP (`n` is TCP, `nu` is UDP).

- `open(host_or_"user@host", "h...", attrs...)` – first argument is a string: opens a new authenticated SSH **session** and, by default, an interactive shell channel on it (section 4.3).

- `open(s, "hc"/"hs", attrs...)` – first argument is an existing SSH session or channel file: opens a **new channel** (`hc`) or SFTP handle (`hs`) on the same underlying session (reusing the transport and authentication, no new handshake). Opening on a channel opens a **sibling** on the session owner, not a child of the channel.

A session opened with `channel=no` has no channel of its own (`Fs_SSH` alone, no `Fs_Socket`). It exists so later `open(s, "hc")` / `open(s, "hs")` calls have an authenticated transport. `channel=yes` is the default. Combining `channel=no` with mode `hc` or `cmd=` is contradictory and fails with a bad-attribute error.

```
s := open("jafar@" || host, "h", "key=" || keypath)
write(s, "ls -l
n")
while line := reads(s) do write(line)
close(s)

s := open("user@host", "h", "key=id_ed25519", "channel=no")
c1 := open(s, "hc", "cmd=uname -a")
c2 := open(s, "hc", "cmd=uptime")
```

4.2 4.2 Host string

The first argument is `host`, `user@host`, or either with `:port`. The port uses the same `host:port` form as socket `open()`; IPv6 uses `[addr]:port`. A bare IPv6 literal is not misparsed: an unbracketed host is split on `:` only when there is a single colon.

`user@host` wins over a `user=` attribute. If neither is given, libssh's default user (the local login name) is used.

4.3 4.3 Default channel: shell, not bare session

`open(host, "h", ...)` with no `cmd=` and without `channel=no` opens a shell on a remote PTY immediately, so the common interactive case is one handle.

The PTY is requested with `TERM` (from `term=`, else `$TERM`, else `xterm`) and a window size (from `cols=` / `rows=`, else the local tty via `TIOCGWINSZ`, else 80x24). Without a real size, remote `ls(1)` and friends fall back to one column.

A shell channel has no clean command boundaries: prompt text, echoed input, and output all interleave. For scripted use, prefer mode `hc` with `cmd=`, which has clean stdout and a real exit status:

```
c := open(s, "hc", "cmd=ls -l")
```

5 5. Attributes and authentication

Trailing `open()` arguments are `name=value` strings. Empty values and unknown names fail (error 1331). Every attribute must contain `=`.

Attribute	Meaning	Notes
`user=`	Remote login name	Overridden by `user@host`
`key=`	Path to a private key file	Same convention as SSL's `key=`
`keypass=`	Passphrase for an encrypted private key	Same name as TLS/crypto; distinct from SSH `password=`
`password=`	Remote login password	
`verifyPeer=`	`yes` (default) or `no`	Same spelling as TLS; overrides mode `-`
`hostkeyfile=`	Path to an OpenSSH `known_hosts` file	Passed to libssh as `SSH_OPTIONS_KNOWNHOSTS`
`cmd=`	Command for a one-shot exec channel	Requires mode `hc`; contradicts `channel=no`
`channel=`	`yes` (default) or `no`	`no` is transport-only

Attribute	Meaning	Notes
`term=`	Remote `TERM` for a PTY	Default `$TERM` or `xterm`
`cols=` / `rows=`	PTY size	Default from the local tty, else 80x24

`keypass=` is the shared name for a key-file passphrase on TLS, crypto, and SSH. SSH `password=` remains the remote login password.

The `-` mode-character flag disables peer verification on TLS (`ne-`) and SSH (`h-`), the same as `verifyPeer=no`. An explicit `verifyPeer=` overrides the mode flag. `Fs_Encrypt` and `Fs_SSH` are mutually exclusive contexts, so there is no conflict.

A leading integer attribute is a connect timeout in milliseconds and is applied as libssh's `SSH_OPTIONS_TIMEOUT` / `TIMEOUT_USEC`. The default port is 22 when `:port` is omitted.

5.1 5.1 Authentication order

If both `key=` and `password=` are given, they are tried in the order the attributes appear in the call, not in a hardcoded priority. If **neither** is given, the implementation falls back to `ssh_userauth_publickey_auto()` – default identities under `~/.ssh` and a running agent – not to password authentication. Agent use through this fallback is supported; an explicit `agent=` attribute and keyboard-interactive auth remain out of scope.

`key=` does not expand `~`. Callers (and the `ussh` demo) must expand a leading `~/` themselves; libssh treats the path literally.

6 6. Examples and use cases

These examples assume a Unicon build with the `secure shell` feature. Connection failures fail with `&errortext` set; the usual form is `s := open(...) | stop(&errortext)`. The packaged demo `uni/progs/ussh.icn` is a small client that covers the interactive and one-shot command cases.

6.1 6.1 Modes

Like every other `open()`, SSH uses the second argument as a mode string (it defaults to `"r"` when omitted, which is not useful for SSH). `h` / `H` is a mode character in that same scan, alongside `r` / `w` / `n` / `e` / `m`. After `h`, `c` selects a command/channel and `s` selects SFTP – the same "facility letter, then modifier" order as `nu` (UDP) and `ms` (Messaging short-request). `n` alone is TCP. Without a preceding `h`, `c` is still create+write. Trailing `"name=value"` arguments are attributes, the same as SSL uses for `key=` and friends.

Mode characters combine. `"h-"` is SSH plus skip host-key verification (the `-` flag also disables SSL peer checks). `"hc"` is a channel (exec when `cmd=` is given, else a shell). `"hs"` is SFTP; `r` / `w` / `a` / `b` set transfer intent (`"hsw"`).

Form	Kind	Handle
`open(host, "h", ...)`	mode `h`	New session plus a PTY shell
`open(host, "h-", ...)`	mode `h` and `-`	Same, no host-key check
`open(host, "hc", "cmd=...")`	mode `hc`, attribute `cmd=`	New session plus an exec channel
`open(host, "h", "channel=no")`	mode `h`, attribute `channel=`	Session only (no channel)
`open(s, "hc")`	mode `hc` on an existing SSH file	Extra shell channel on that session
`open(s, "hc", "cmd=...")`	mode `hc`, attribute `cmd=`	Extra exec channel on that session
`open(s, "hsw", "path=...")`	mode `hs` plus access chars	SFTP file or directory
`open(host, "hsw", "path=...")`	mode `hs` on a host string	New session plus SFTP (one handle)
leading integer	attribute (timeout)	Connect timeout in milliseconds

`channel=no` with `hc` or `cmd=` fails (error 1331). `cmd=` without `c` also fails.

6.2 6.2 Interactive shell

With no `cmd=`, `open(..., "h")` requests a remote PTY and shell. `net::ssh_interactive(s)` attaches the local tty (raw mode, `select()` with `&input`,

CRLF for display):

```
import net

procedure main()
   s := open("user@host", "h", "key=/home/user/.ssh/id_ed25519") |
        stop(&errortext)
   ssh_interactive(s)
   close(s)
end
```

`term=`, `cols=`, and `rows=` override the PTY. Mode `"h-"` skips `known_hosts` (useful on a first connect before the key is installed). Password auth uses `password=` instead of, or in addition to, `key=`.

6.3 6.3 Remote command

Mode `"hc"` with `cmd=` runs one command and returns its stdout through ordinary `read()` / `reads()`. `c["exitstatus"]` is the remote exit code once it has arrived:

```
procedure main()
   s := open("user@host", "hc", "key=/home/user/.ssh/id_ed25519",
             "cmd=uname -a") | stop(&errortext)
   while write(read(s))
   write("exit status: ", s["exitstatus"])
   close(s)
end
```

The host string may include a port (`user@host:2222`) or an IPv6 address (`user@[2001:db8::1]:22`). A leading integer is a connect timeout in milliseconds:

```
s := open("user@host", "hc", 5000, "cmd=true") | stop(&errortext)
```

6.4 6.4 Ordered stdout and stderr

Stream `reads()` see stdout only. When stdout and stderr must stay in arrival order, or when stderr should go to `&errout`, use `receive()`:

```
s := open("user@host", "hc", "key=/home/user/.ssh/id_ed25519",
          "cmd=echo out; echo err 1>&2; exit 3") | stop(&errortext)
while m := receive(s) do
   case m.addr of {
      "stdout" : writes(&output, m.msg)
      "stderr" : writes(&errout, m.msg)
      "exit"   : write(&errout, "[exit ", trim(m.msg), "]")
      }
close(s)
```

`c["stderr"]` is the other stderr path: it peeks the current accumulated buffer, without preserving order against stdout. See section 8.

6.5 6.5 Several channels on one session

`channel=no` opens an authenticated transport. Later `open(s, "hc")` / `open(s, "hs")` calls add exec, shell, or SFTP handles on that session. Closing `s` invalidates every derived handle:

```
s := open("user@host", "h", "key=/home/user/.ssh/id_ed25519",
          "channel=no") | stop(&errortext)
c1 := open(s, "hc", "cmd=uname -a") | stop(&errortext)
c2 := open(s, "hc", "cmd=uptime") | stop(&errortext)
```

```
while write(read(c1))
while write(read(c2))
close(c1)
close(c2)
close(s)
```

Opening on an existing channel opens a sibling on the session owner, not a child of that channel.

6.6 6.6 SFTP

The same session carries SFTP. Mode `"hs"` plus `path=` opens a remote file or, if the path is a directory, lists it. `stat`, `remove`, and `rename` take the session as a leading argument:

```
s := open("user@host", "h", "key=/home/user/.ssh/id_ed25519",
          "channel=no") | stop(&errortext)

c := open(s, "hs", "path=/tmp/out.bin", "w") | stop(&errortext)
writes(c, ReadBytes("local.bin"))
close(c)

info := stat(s, "/tmp/out.bin") | stop(&errortext)
write("remote size: ", info.size)

d := open(s, "hs", "path=/tmp") | stop(&errortext)
while write(read(d))
close(d)

rename(s, "/tmp/out.bin", "/tmp/out.bin.bak")
remove(s, "/tmp/out.bin.bak")
close(s)
```

6.7 6.7 Concurrent hosts

Each connection is blocking. Concurrent hosts use Unicon `thread`, one thread per in-flight session:

```
procedure run_cmd(host, key)
   local s, line
   s := open(host, "hc", "key=" || key, "cmd=uname -a") | {
      write(&errout, host, ": ", &errortext)
      fail
      }
   while line := read(s) do write(host, ": ", line)
   close(s)
end

procedure main()
   local host, key
   key := "/home/user/.ssh/id_ed25519"
   every host := "user@alpha" | "user@beta" | "user@gamma" do
      thread run_cmd(host, key)
end
```

Channels that share a session serialize on the session mutex (section 10.1). Separate sessions do not.

7 7. Error handling

Connection and channel failures **fail**, they do not `runerr`. This matches the general `open()` pattern: set `&errortext`, tear down what was allocated, and fail, so callers write `if s := open(...) then ... else ...`. Runtime errors are reserved for type mistakes (passing a non-SSH file to `open(s, "hc")`, a bad peek name on an SSH file).

Error numbers, defined in `src/runtime/data.r`:

Number	Text	Typical cause
1330	SSH error	Connect failure, allocation, bad host string
1331	bad ssh attribute	Unknown name, empty value, `cmd=` without `c`, `channel=no` with `hc`
1332	SSH authentication error	No method succeeded
1333	SSH host key verification error	Server not in `known_hosts` (when verifying)
1334	SSH channel error	Channel open, I/O, or a closed/cascaded handle
1335	SFTP error	SFTP subsystem, path, or transfer failure

`set_ssh_errortext()` appends libssh's own message where available. Blocking libssh calls run under `DEC_NARTHREADS`; `&errortext` is set only after the thread is re-registered, because Unicon allocation is not allowed while unregistered.

8 8. I/O: streams, peek fields, and `receive()`

Three tiers, layered simple-default to full-fidelity. All three share one per-channel event queue populated by libssh callbacks (section 8.4).

8.1 8.1 Plain stream I/O

`read()` / `write()` / `reads()` / `writes()` work unmodified via the existing `u_read()` / `u_write()` dispatch, extended with an `Fs_SSH` branch. Stream reads consume **stdout only**. They are enough when the caller does not need to distinguish stderr or preserve cross-stream order.

Writes to a channel call `ssh_channel_write()` and retry short writes. Writes to a transport-only session (no channel) fail with

1. SFTP regular files use `sftp_write()` on the same path.

8.2 8.2 Peek fields (`[ ]` and `key()`)

Status on SSH files is a non-destructive peek through `[ ]`. Unknown names raise 1331. An unpopulated field fails. Boolean fields that answered succeed with `"yes"` or `&null`. `key(c)` generates every answerable field, including false booleans. `c["*"]` snapshots those fields under one lock. Peeking a closed handle is error 174; `close()` keeps the original name so `image(c)` remains `file(user@host)`. `key(c)` that has not yet produced a name also raises 174 if the handle is already closed. After the first suspend, Similar to messaging: `!` generates content, `key()` generates names. Channels keep line-generating `!` (they carry `Fs_Socket`). Transport-only sessions (`channel=no`) are not line streams; `!s` fails.

A few fields are lists: one item is still a list of length 1. Walk members with `!s["authmethods"]`.

Channel (exec, shell, or SFTP handle). `c["type"]` is `channel` or `sftp`.

Name	Value
`type`	`channel` or `sftp`
`exitstatus`	Integer remote exit code. **Fails until the exit-status
`stderr`	Accumulated stderr (a peek, not a drain). Fails if empty.
`eof`	`"yes"` once the remote sent EOF, else `&null`.
`bytesread` / `byteswritten`	Running counters (integers).

Session (the authenticated connection, including `channel=no`).

Name	Value
`type`	`session`
`fingerprint`	Server host-key SHA-256, e.g. `SHA256:x3Gnt...`

Name	Value
`authmethods`	List of method names (`publickey`, `password`,
`cipher`	Negotiated inbound cipher:
`kex`	Key-exchange algorithm: `curve25519-sha256`,
`mac`	Inbound MAC: `hmac-sha2-256`, `hmac-sha2-512`
`serverbanner`	Server identification string, e.g. `SSH-2.0-OpenSSH_9.6`
`connected`	`"yes"` or `&null`

```
s := open("user@host", "h", "key=id_rsa", "channel=no") |
   stop(&errortext)
write("type=", s["type"], " kex=", s["kex"])
every m := !s["authmethods"] do
   write("auth ", m)
every k := key(s) do
   write("session field ", k)

c := open(s, "hc", "cmd=apply-config /tmp/newconfig.json")
while line := reads(c) do
   process(line)
if
c["exitstatus"] = 0 then
   write("ok")
else
   write("failed: " || (c["stderr"] | ""))
every k := key(c) do
   write("channel field ", k)
close(c)
close(s)
```

Preferring callback state over libssh's blocking `get_exit_*` helpers avoids version-sensitive fallbacks once callbacks are registered. Failing until the status is ready reuses the ordinary Unicon "fail = not ready" idiom, the

same as `read()` failing at EOF.

8.3 8.3 `receive()` for ordered events

A command can produce interleaved stdout and stderr where the caller needs to know *when* an stderr message occurred relative to stdout. Neither `reads()` (stdout only) nor `c["stderr"]` (a separate peek) can preserve that.

`receive(c)` on an SSH channel yields the next queued event as a `posix_message` whose `addr` is `"stdout"`, `"stderr"`, or `"exit"` and whose `msg` is the payload. The exit status is kept as a **string** so `receive()`'s return type stays uniform across every kind of file it can be called on; Unicon's `=` already coerces numeric strings.

```
c := open(s, "hc", "cmd=process_records")
while m := receive(c) do
   case m.addr of {
      "stdout" : write("OUT: " || m.msg)
      "stderr" : write("ERR: " || m.msg)
      "exit"   : write("done, status " || m.msg)
      }
```

`receive()` fails once the channel is exhausted.

8.4 8.4 Arrival-order queue

Preserving cross-stream order requires libssh's **callback API** (`ssh_channel_callbacks`, with a `channel_data_function` receiving an `is_stderr` flag) – not `ssh_channel_poll()` / `ssh_channel_read()`, which buffer stdout and stderr separately and lose interleaving. Callbacks are registered **before** the channel is opened so early data cannot land in libssh's own buffers.

Each callback appends a tagged chunk (`SSH_CHUNK_STDOUT` / `STDERR` / `EXIT`) to a heap-allocated linked list on the `SSHfile`. Callbacks can fire inside blocking libssh calls made while the thread is unregistered (`DEC_NARTHREADS`), where Unicon allocation is not allowed. `ssh_pump()` drives `ssh_channel_poll_timeout()` so the callbacks run. Stream reads consume

stdout chunks; `c["stderr"]` peeks stderr chunks; `receive()` pops whatever is at the head. One mechanism, three accessors.

9 9. Interactive shells and `select()`

9.1 9.1 Remote PTY

An interactive channel requests a PTY before `shell`. Size and `TERM` are described in section 4.3. The session file is also marked `Fs_Socket` so existing socket dispatch (including `select()` and `get_fd()`) applies.

9.2 9.2 Partial-line `reads()`

Interactive prompts have no trailing newline. The original socket line helper (`sock_getstrg`) waits for `\n` and hung on a banner or prompt. SSH therefore has its own `ssh_getstrg()`: it reads stdout bytes from the queue and returns a partial line at EOF or when the caller's buffer fills, without blocking for a newline that will never come. A seen-but- unconsumed newline is remembered in `nl_pending` (libssh has no `MSG_PEEK`). `reads(s, n)` on a channel is a byte read from the same queue and is the right primitive for an interactive loop.

9.3 9.3 `select()` readiness

`select()` on an SSH file must not hang when libssh already buffered the banner during `open()`. `ssh_file_pending()` nonblocking-pumps the session and reports ready when stdout (or EOF) is queued. Only stdout matters here: stream `reads()` do not consume stderr/exit chunks, so a nonempty queue of those alone must not make `select()` claim the file is readable. `get_fd()` returns `ssh_get_fd(session)` so the kernel fd participates in the same `select()` set as ordinary sockets.

On Windows, `select(&input)` and `Attrib(f, "tty=raw")` / `"tty=sane"` were added so an interactive client can mux the console with the channel on both Unix and Windows console `iconx`. Those tty attributes are general (they apply to `&input`), not SSH-specific.

9.4 9.4 net::ssh_interactive() and ussh

uni/lib/ssh.icn provides ssh_interactive(s): put the local tty in raw mode, mux &input with the remote channel via select(), map lone LF to CRLF when copying remote output to a raw local terminal (needed because raw mode disables ONLCR), and restore the tty before returning. It does not close s. The ussh demo (uni/progs/ussh.icn) is a small OpenSSH-like client built with the other uni/progs demos: interactive shell, or a one-shot remote command via receive().

```
s := open("user@host", "h", "key=...") | stop(&errortext)
ssh_interactive(s)
close(s)
```

10 10. Concurrency and channel lifecycle

10.1 10.1 Shared mutex

Every b_file already has a mutexid, and all I/O in fsys.r already locks around it. When a channel is created via open(s, "hc") / open(s, "hs"), it does **not** allocate a fresh mutex id – it copies the session's existing mutexid. Every channel sharing a session then serializes through the same lock via the existing locking calls in u_read / u_write / receive / Attrib. libssh sessions are not safe for uncoordinated concurrent access; two threads each holding a different channel on the same session block each other during actual I/O because they contend for the same mutex.

10.2 10.2 Child tracking and cascade close

A Unicon list of b_file pointers was considered so the moving collector would relocate them. SSHfile is malloc'd and therefore does not move, so the session owner holds a C linked list of child SSHfile * (children / next / parent). No Unicon list is required.

close(s) on the session walks that list, force-closes every remaining channel and SFTP handle (same close-hook logic as an explicit close()), marks

each child `closed`, clears its queue, and then disconnects and frees the session. The child's Unicon `b_file` stays alive until its own `close()`, but is unusable immediately – no dangling libssh objects, no readable leftovers. Closing a channel unlinks it from the owner and frees only that channel.

The SFTP subsystem is created lazily, once, on the session owner (`ssh_owner_sftp()`) and shared by every SFTP file or directory on that session. It is freed when the session closes.

10.3 10.3 Blocking model

I/O uses blocking libssh calls plus Unicon's existing `thread` mechanism – one thread per in-flight host or channel. Chosen over non-blocking libssh plus `select()` as the *primary* model: it requires no partial-read/retry state machines. `select()` readiness (section 9.3) is implemented so interactive muxing works; it is not a full non-blocking I/O API.

11 11. SFTP

Three shapes, each mapped onto an existing Unicon pattern.

11.1 11.1 File transfer – `read()` / `write()`

```
c := open(s, "hs", "path=/tmp/firmware.bin", "w")
while writes(c, ReadBytes(local_fw))
close(c)
```

Mode `"hs"` is SFTP; read/write/append intent comes from the usual mode characters (`r` / `w` / `a` / `b`), either in the same string (`"hsw"`) or as a trailing mode-only token. `path=` is required. Backed by `sftp_open()` / `sftp_read()` / `sftp_write()` / `sftp_close()`. Structurally a plain file – another variant feeding the existing dispatch. SFTP regular files are **not** marked `Fs_Socket`: they are byte streams, not select-able channels.

11.2 11.2 Directory listing

Local `open()` on a path that turns out to be a directory transparently switches to `opendir()` / `readdir()`, sets `Fs_Directory`, and `reads()` yields one entry name per call. SFTP does the same: `open(s, "hs", "path=/tmp/configs")` checks the remote path with one `sftp_stat()`; if it is a directory, `sftp_opendir()` / `sftp_readdir()` provide the same `Fs_Directory` behavior.

11.3 11.3 Metadata – `stat()`, `remove()`, `rename()`

These extend existing builtins via the leading-optional-file argument idiom, rather than introducing `sftp_remove()` names. The signature change is backward-compatible: a string first argument is still the local-filesystem operation.

- `remove(s, path)` – `sftp_unlink` when `s` is an SSH session or channel file.
- `rename(s, from, to)` – `sftp_rename`. `rename` gained a third argument for this form; `rename(s1, s2)` is unchanged.
- `stat(c)` – `sftp_fstat` on an already-open SFTP file.
- `stat(s, path)` – a single `sftp_stat` request, without paying for `sftp_open()` + `sftp_fstat()`. Both entry points are worth supporting rather than just one.

```
if info := stat(s, "/tmp/firmware.bin") then
   if info.size = expected_size then pull_it()
```

`sftp2rec()` populates the same `posix_stat` record as `stat2rec()`. SFTP's attribute set is protocol-version dependent; fields the server did not send are left **null** rather than reported as zero, so callers can tell "absent" from "genuinely zero." SFTP carries atime/mtime, not ctime. Size, uid/gid (name or numeric), permissions-as-mode-string, and file type (`d` / `l` / `c`) are filled when the corresponding flags are present.

12 12. Build integration

`configure --disable-ssh` turns the feature off. Otherwise `CHECK_LIBSSH` in `aclocal.m4` probes for `<libssh/libssh.h>` and `ssh_new`, setting `HAVE_LIBSSH`. The probe looks under `/usr/local` (FreeBSD) and `/opt/homebrew` (macOS) when those prefixes contain the headers, so later compiles do not miss `libssh/libssh.h`. `--with-libssh=DIR` overrides the prefix. A required `--enable-ssh` without the library is a configure error; optional (default) absence degrades gracefully, as with other optional libraries.

`&features` reports `secure shell` when the library was found. `tests/posix/Makefile` skips `ssh` when the feature is absent. CI installs `libssh-dev` (or the platform equivalent) so the support builds and the deterministic half of the test runs. README install notes list `libssh-dev` / `libssh-devel` / `brew install libssh` alongside the other optional libraries.

Guards are `#if HAVE_LIBSSH` throughout `rstructs.h`, `rmacros.h`, `feature.h`, `sys.h`, `fsys.r`, `rposix.r`, `fxposix.ri`, and `fmisc.r`.

13 13. Status and remaining work

Keyboard-interactive authentication and an explicit agent attribute. The `publickey_auto` fallback already talks to a running agent.

Non-blocking I/O as the primary concurrency model. `select()` readiness is implemented; the I/O calls themselves still block. A `ssh_set_blocking(sess, 0)` plus resume state machine is the obvious next step if thread-count pressure appears.

SSH server. libssh can do it; nothing in the runtime exposes it. The mode character and file-status bit leave room, but the accept/listen path is not designed.

Default channel type. Shell-by-default shipped for mode `"h"`. Scripted use uses `"hc"` with `cmd=`. Reconsidering the default remains open and would be a compatibility break.

Timeouts. TCP-connect timeout is the leading integer attribute. There is no separate handshake+auth timeout beyond what that covers, and no exec timeout – a stuck command blocks until `close()`, matching ordinary blocking sockets.

Trust-on-first-use. Verification is check-only (`ssh_session_is_known_server`). A new host fails with 1333 unless the caller uses `h-` or installs the

key in `known_hosts` out of band. The runtime does not write the store.

`~` **in** `key=`**.** Not expanded. The helper and `ussh` expand `~/` themselves.

14 Appendix: test coverage

`tests/posix/ssh.icn` is deterministic without a server: it checks that `&features` reports `secure shell`, and that bad attributes, `channel=maybe`, `channel=no` combined with `hc`/`cmd=`, `cmd=` without mode `c`, and a connect to a closed local port all fail with `&errortext` set. A live round-trip runs only when `UNICON_SSH_TESTHOST` is set (host or `user@host`), with optional `UNICON_SSH_TESTPORT`, `UNICON_SSH_TESTKEY`, `UNICON_SSH_KNOWNHOSTS`, and `UNICON_SSH_SFTPPATH`. The live path covers exec stdout, `c["exitstatus"]`, `receive()` event tags, partial-line `reads()` of a `printf` with no newline, and SFTP write/read/stat/remove of a binary payload.

`tests/posix/Makefile` skips the test when the feature is absent. Expected default output is `tests/posix/stand/ssh.std`.

15 References

References

[1] Clinton Jeffery. "How to Write a Unicon Technical Report". doc/utr/utr15.tex. 2013.

[2] The libssh Project. "libssh – The SSH Library". https://www.libssh.org/. 2026.

[3] Clinton Jeffery. "The Unicon Messaging Facilities". doc/utr/utr13.odt.