Linux ATM internal signaling protocol
Version 0.2

Werner Almesberger
werner.almesberger@lrc.di.epfl.ch

Laboratoire de Réseaux de Communication (LRC)
EPFL, CH-1015 Lausanne, Switzerland

November 5, 1996

Contents

1 Introduction

The Linux kernel only implements a very simple protocol to support ATM signaling. All the complexity of ATM signaling is delegated to a user-mode demon process. [1] Figure 1 illustrates the configuration. This document describes the protocol that is used for internal communication between the kernel and that demon process.[1] Note that this protocol is steadily evolving, as described in appendix D.

The internal signaling protocol is much simpler than the signaling protocol that is used on the network (i.e. Q.2931 or a derivate thereof, [2]), because the following assumptions can be made:

- communication is well-behaved (reliable, preserves sequence, etc.)
- the communicating parties agree on every protocol detail, including the version or revision of the protocol
- the communicating parties always cooperate
- both parties share the same architecture

Well-behaved communication obviates the need for checksums, protocol timers, retransmissions, re-sequencing, etc. Agreement on the protocol eliminates the need for supporting compatibility modes, feature discovery mechanisms, etc. Cooperation simplifies certain mapping operations, e.g. it is even safe to exchange pointers to kernel data structures. Finally, sharing the same architecture avoids the need for machine-independent data formats and the corresponding coding procedures.

2 System overview

The following sections give an overview of the framework the internal signaling protocol operates in.

2.1 Communication mechanisms

As mentioned above, communication between the kernel and the signaling demon is generally assumed to be reliable and to preserve the sequence of messages. Furthermore, it is assumed that there is a queue when sending to the demon, so communication is asynchronous. (See also figure 2.)

Communication is synchronous from the signaling demon to the kernel, although the protocol design would also allow for the use of an asynchronous mechanism. There is only a single sequential communication path in either direction. Messages related to different sockets may be interleaved.

2.2 State information

The kernel and the demon both keep state information (i.e. descriptors) for sockets and connections. The state information in the demon does not always correspond to the existence of a socket owned by a user process, e.g. no notification is sent to the demon when executing a `socket` system call, so the demon doesn't know about the socket at that time.

When removing a socket, the kernel must notify the demon ("de-register") if state information has previously been created (if it has "registered" the socket). The demon de-registers a socket automatically in certain error cases.

The moment after which the kernel no longer has to expect a message for a given socket is well defined. This allows the use of pointers to kernel data structures in the internal protocol. Note that the reverse is

[1] The signaling demon is not directly involved in opening VCs for data traffic. All that is done by the kernel.

not true: the demon may receive messages from the kernel even when the demon has already discarded all information associated with the socket.

2.3 Socket model

The protocol is designed for the use with Berkeley-style sockets. All the general operations, such as binding to a local address, requesting an outgoing call (either blocking or non-blocking), accepting of incoming connections, etc. are supported.

One particularity is that the `bind` operation does not affect the state of the socket in the demon: In the socket API, the `bind` operation normally (e.g. for INET domain sockets) verifies that the requested service access point (SAP; e.g. address and port) is valid and then reserves that SAP for exclusive use by that socket.

With ATM, the actual binding and reservation of a local service access point (SAP) is deferred until a `listen` request is sent. This is necessary, because on ATM, one needs to distinguish between binding to specify the caller's local address and binding to specify a SAP for attribution of incoming connections. This is a distinction that doesn't exist in the traditional socket API semantics.

As illustrated in figure 3, incoming connections are queued on the listening socket on both sides.[2] They have to be processed strictly in sequence, because they don't have a unique identification at this time and are therefore referenced by specifying the listening socket. See also section 4.1.3.

3 Messages flows

This section describes the message types, the message semantics, and gives a textual description of the protocol behaviour. Please see appendix A for flow diagrams of either party.

3.1 Message types

The message types are defined in `/usr/include/linux/atmsvc.h`:

```
enum atmsvc_msg_type { as_catch_null,as_bind,as_connect,as_accept,as_reject,
  as_listen,as_okay,as_error,as_indicate,as_close,as_itf_notify,
  as_modify,as_identify,as_terminate };
```

`as_catch_null` is used only to detect programming errors and must never appear in an actual message. `as_itf_notify` is not directly related to connection handling and is described in section 5.1. The other message types are described below.

3.2 `as_okay`

The `as_okay` message is sent by the demon to acknowledge successful processing of the previous message that was received from the kernel. Depending on the operation, `as_okay` may return various parameters to the kernel, see table 1.

[2]The socket descriptors in the kernel are drawn with a horizontal separating line to illustrate that they actually consist of a general socket structure and the ATM-specific VC structure. There are cases where the VC structure is used without its own socket structure, e.g. for Arequipa [4]. The internal signaling protocol references only the VC part.

3.3 `as_error`

The `as_error` message is sent by the demon to inform the kernel that an error has occurred while processing the previous kernel message (for that socket). `as_error` is not acknowledged. The demon destroys all internal context for a connection when sending an `as_error`.

Note that the demon may receive an `as_close` after emitting the `as_error` (i.e. at a time when the demon has already forgotten about the connection), so it has to ignore `as_close` messages which are received for an unknown connection. (See also figure 10.)

3.4 `as_close`

The `as_close` message can be used for three purposes:

- kernel to demon: to indicate normal closing of the connection by the application, e.g. with the `close` system call
- demon to kernel: to indicate normal closing of the connection by the remote party
- demon to kernel: to indicate abnormal closing of the connection, e.g. by the network

From the kernel's perspective, an `as_close` message it sends is always acknowledged with exactly one `as_close` or `as_error` message. If the demon sent an `as_close` or `as_error` message before receiving the kernel's `as_close` message, no further message is generated by the demon.

The demon destroys all state information about a socket or connection in the following cases:

- when acknowledging an `as_close` message
- when receiving an `as_close` message after an `as_close` message was emitted

Likewise, the kernel acknowledges all `as_close` messages from the demon, except when the kernel has already sent an `as_close` or `as_error` message.

3.5 `as_bind`

The `as_bind` message is sent by the kernel (e.g. in a `bind` system call) to verify if the local address chosen by the application is acceptable. Note that no state information is created in the demon when handling an `as_bind` message. It is therefore not necessary to notify the demon of removal of a socket if only an `as_bind` message was sent for it.

The demon replies with either an `as_okay` or an `as_error` message. The kernel must not send any other messages for this socket while waiting for the response.

3.6 `as_connect`

The `as_connect` message is sent by the kernel to request establishment of an outbound connection (e.g. in a `connect` system call).

The demon responds either with `as_okay` or with `as_error`. Note that the client is allowed to close the connection (by sending `as_close` and awaiting the confirmation) before the `as_connect` is acknowledged.

The demon creates state information (i.e. it allocates a socket descriptor) on reception of an `as_connect` message.

3.7 as_listen

The as_listen message is sent by the kernel to register a service access point (SAP) for incoming calls, e.g. in a listen system call. The demon replies either with as_okay or with as_error. The kernel must not send any messages for this socket before receiving a response.

If the listen operation succeeds, state is created in the demon and the kernel must de-register the socket.

Note that the demon cannot force the kernel to close a listening socket.[3]

3.8 as_indicate

An as_indicate message is sent to the kernel when an incoming call is received for the corresponding SAP. The kernel can respond to an as_indicate either with an as_accept message (see section 3.9) or with an as_reject message. as_indicate messages must be answered in the order in which they arrive.

Note that, because each as_indicate message has to be answered, the listen queue of a listening socket can only be emptied with certainty after that socket has been de-registered and after the demon's acknowledgement has been received. The kernel may still receive as_indicate messages before that.

3.9 as_accept

The as_accept message is sent by the kernel to accept an incoming connection that has previously been announced by the demon with as_indicate (e.g. in an accept system call).

The demon responds with as_okay or with as_error. Note that the client is allowed to interrupt the operation (by sending as_close and awaiting the confirmation) before the as_accept is acknowledged.

When receiving an as_accept message, the already present socket descriptor is used.

3.10 as_reject

The as_reject message is sent by the kernel to reject an incoming connection that has previously been announced by the demon with as_indicate.

Unlike as_close, as_reject is not confirmed by the signaling demon. The kernel can destroy all internal context for a connection when sending an as_reject.

4 Message contents

This section describes message contents and which fields have to be set in which messages. Note that the representation of all message contents is machine-specific.

4.1 Message fields

struct atmsvc_msg, the message structure used in both directions for the internal signaling protocol is declared in /usr/include/linux/atmsvc.h:

[3]Such functionality may appear to be desirable for starting clean-up procedures when imminent failure of the signaling subsystem is indicated. The current implementation uses a different approach: all sockets are marked as unavailable internally by the kernel when the special control socket for signaling messages is closed, which happens automatically when the signaling demon terminates. All subsequent operations on such sockets (except for close) fail and errors are returned to the application.

```
struct atmsvc_msg {
        enum atmsvc_msg_type type;
        atm_kptr_t vcc;
        atm_kptr_t listen_vcc;
        int reply;
        struct sockaddr_atmpvc pvc;
        struct sockaddr_atmsvc local;
        struct atm_qos qos;
        struct atm_sap sap;
        unsigned int session;
        struct sockaddr_atmsvc svc;
} __ATM_API_ALIGN;
 * Message contents: see ftp://icaftp.epfl.ch/pub/linux/atm/docs/isp-*.tar.gz
 */
 * Some policy stuff for atmsigd and for net/atm/svc.c. Both have to agree on
 * what PCR is used to request bandwidth from the device driver. net/atm/svc.c
 * tries to do better than that, but only if there's no routing decision (i.e.
 * if signaling only uses one ATM interface).
 */
#define SELECT_TOP_PCR(tp) ((tp).pcr ? (tp).pcr : \
  (tp).max_pcr && (tp).max_pcr != ATM_MAX_PCR ? (tp).max_pcr : \
  (tp).min_pcr ? (tp).min_pcr : ATM_MAX_PCR)
#endif
```

The fields are described in the following sections. Note that the message has variable length, see section 4.1.10.

Generally, if a field is included in a message (see section 4.2), its content and the content of any subordinate fields must be correct. The only exception are cases where the kernel is allowed to forward arguments passed from the application to the signaling demon without previously checking their coding. Those exceptions are indicated where applicable.

4.1.1 `type`

The `type` field carries the message type (see also section 3.1). It is required in every message.

4.1.2 `vcc`

This field uniquely identifies the SVC socket to which the message applies. `vcc` values are assigned by the kernel.

When sent to the kernel, `vcc` is guaranteed to reference a socket that still exists. In the opposite direction, `vcc` may reference a socket that no longer exists, but it will never reference a different existing socket, i.e. uniqueness is still guaranteed.

`vcc` is used in most messages, with the exception noted in section 4.1.3.

4.1.3 `listen_vcc`

This field is used in the messages `as_indicate`, `as_accept`, and `as_reject`. It is the unique identifier of the listening socket to which the incoming connection is attributed.

In `as_indicate` and `as_reject` messages, `listen_vcc` is the only means to identify the corresponding socket, because no unique identifier has been assigned to the connection. The socket to which an `as_accept` message applies is also determined by `listen_vcc`. The `vcc` field contained in the message is then used as the identifier of the new socket.

4.1.4 `reply`

This field is used in `as_close` messages from the demon to the kernel, in `as_error` messages, and in `as_reject` messages. If zero (`as_close` only), it indicates normal closing, e.g. when the remote party has closed the connection. If negative, it is the error code describing the reason why the operation has failed or why the connection was interrupted.

4.1.5 `aal`

`aal` carries the AAL number in `as_bind`, `as_connect`, and `as_listen` messages. This field is necessary, because the SAP part of an address structure does not contain the AAL.[4]

Note that `aal` isn't conveyed in `as_indicate`, although that message contains SAP information too. The reason for this is that the AAL is already implied by the listening socket and therefore doesn't need to be repeated.

Note that kernel and signaling demon may not support the same set of AALs. Both need therefore to check AAL numbers received from the respective other side.

4.1.6 `pvc`

This field contains the interface and the network-assigned connection identifier on which the SVC has to be created. It is sent to the kernel as soon as the information becomes available, i.e. in the `as_okay` after `as_connect`, and in `as_indicate`.

4.1.7 `local`

This field contains the local address, but no SAP information.[5]

In an `as_connect` message, it indicates the address that should be used as the "calling party number". The signaling demon will try to assign a valid address if a null address is passed. In the `as_okay` message that is sent to the kernel after a successful connection setup, that address is returned.

Likewise, `as_okay` after accepting an incoming connection contains the settings to use for the newly created socket.

Note that the content of this field may be invalid in messages from the kernel to the signaling demon.

4.1.8 `qos`

In an `as_connect` message, this field contains the Quality of Service parameters requested by the application. In the corresponding `as_okay`, it indicates what has been granted by the network and by the remote party (and what therefore should be allocated at the local end).

In `as_listen`, `qos` determines what is acceptable on the SAP. Finally, in `as_indicate`, it specifies what parameters the VC should be created with.

Note that the content of this field may be invalid in messages from the kernel to the signaling demon.

[4] The AAL information may be added to the address structure in the future, so this field may become obsolete.

[5] The fields `svc` and `blli` are used for SAP information, see below.

4.1.9 svc

Like local, this field contains an ATM address. The difference between them is that svc may – in combination with blli – carry a complete SAP specification, with high and low layer information. Note that svc may also be used for local addresses, e.g. in as_bind.

In as_bind, the complete SAP is passed for verification. If as_okay is sent in response to as_bind, the SAP is returned. The returned SAP may differ from the requested one, e.g. the demon may return one of the local addresses if a null address was passed.[6]

Unlike the socket API, the internal signaling protocol requires that the local address (and, for as_listen also the other parts of the SAP) has to be passed in as_connect and as_listen messages, because as_bind does not store that information.

In as_indicate, svc and blli contain the address of the calling party and the connection parameters it requested.

Note that the content of this field may be invalid in messages from the kernel to the signaling demon.

4.1.10 blli

This field is of variable length. If no BLLI information needs to be passed, it is not sent with the message, i.e. the length of the message is `sizeof(struct atmsvc_msg)-sizeof(struct atm_blli)`. If BLLI information is passed, the corresponding number of BLLI descriptors is appended to the message. The number of available BLLI descriptors can be derived from the message size as follows: `bllis = (msg_size-(int) sizeof(struct atmsvc_msg))/(int) sizeof(struct atm_blli)+1;` The pointers in struct atmsvc_msg and in the BLLI fields are not used and may contain undefined values.

blli can be used wherever svc is included.

Note that the content of BLLI descriptors may be invalid in messages from the kernel to the signaling demon.

4.2 Field usage

The type field is included in all messages. The other fields are used as described in table 1.

K→D means that the message is sent from the kernel to the signaling demon. **D→K** denotes the opposite direction.

as_okay messages are always listed below the messages to which they are responses.

5 Other messages

This section describes messages that are not used to control individual connections.

5.1 as_itf_notify

as_itf_notify is sent by the kernel whenever the local address list of an ATM interface has changed. The field pvc.sap_addr.itf contains the interface number.

Note that no other fields (besides the unavoidable type) have valid values in an as_itf_notify message. In particular, no attempt should be made to look up a socket by the vcc or listen_vcc fields.

[6]This is less useful as it may seem, because the local address may change until, for example, the address returned by as_bind is used in an as_listen message.

Message	Direction	`vcc`	`listen _vcc`	`reply`	`aal`	`pvc`	`local`	`qos`	`svc blli`
`as_bind`	**K→D**	●	·	·	●	·	·	·	●
`okay`	**D→K**	●	·	·	·	·	·	·	●
`as_connect`	**K→D**	●	·	·	●	·	●	●	●
`as_okay`	**D→K**	●	·	·	·	●	●	●	·
`as_listen`	**K→D**	●	·	·	●	·	·	●	●
`as_okay`	**D→K**	●	·	·	·	·	·	·	·
`as_indicate`	**D→K**	·	●	·	·	●	·	●	●
`as_accept`	**K→D**	●	●	·	·	·	·	·	·
`as_okay`	**D→K**	●	·	·	·	·	●	·	·
`as_reject`	**K→D**	·	●	●	·	·	·	·	·
`as_error`	**D→K**	●	·	●	·	·	·	·	·
`as_close`	**K→D**	●	·	·	·	·	·	·	·
`as_close`	**D→K**	●	·	●	·	·	·	·	·

Table 1: Presence requirements for message fields.

A Flow diagrams

This appendix contains flow diagrams for kernel and signaling demon behaviour. The representation is loosely based on graphical SDL [5]. Only operations relevant to the protocol are shown. These diagrams are derived from a specification written in Promela [6], which is described in appendix C.

Each diagram shows the kernel side on the left and the signaling demon side on the right. For simplicity, the `as_` prefix has been omitted from message names. Note that the protocol semantics differ from the semantics at the socket API at some places. The most significant difference is that a `bind` operation does not create any state information in the demon. It is therefore valid to issue a `listen` without a prior `bind`. The kernel can also close a socket after a `bind` without notifying the signaling demon (this is not shown in the diagrams).

Figure 4 shows the flows for an active open. The "stray `close`" mentioned on the demon side is a `close` message that was sent by the kernel before receiving the `error`, see also figure 10. Since no synchronization is necessary at the demon side (and uniqueness of VCC descriptor addresses is still guaranteed), this case is simply handled by ignoring `close` messages that are received for unknown sockets.

Figure 5 shows the procedures for listening for incoming calls. Note that the invocations of `k_queuer`, `k_conn`, `k_reject`, and `d_conn` are creations of processes that execute concurrently and asynchronously to their respective parents. Also note that, in order to handle incoming connections in sequence, the messages sent before the creation of these processes must be generated by the listening process.[7]

`k_queuer` picks up all incoming `indicate` messages and transfers them to an internal queue, from which they are later retrieved by the listening socket. In an actual implementation,[8] the functionality of that process would be integrated into the entity that demultiplexes the incoming message stream. `k_queuer` could be stopped after the final `close` has been received.

Note that a similar FIFO has to exist also in the signaling demon. For simplicity, the behaviour of that FIFO is made implicit in the diagrams and in the Promela programs.

[7]If the asynchronous process generated those messages, there would be a race condition with other such asynchronous processes, which may be created at a later time, but which may be scheduled earlier.

[8]E.g. in the current kernel code and also in the Promela programs described in appendix C.

Finally, figure 6 shows the flows per incoming connection. As for outgoing connections, a stray `close` may have to be ignored at the demon side after an `error`.

B Message sequence charts

The message sequence charts (MSCs) in this section illustrate common message flows. In the first four examples, only a single socket is used, so the message destination doesn't need to be indicated. In the last example, a new socket is created, and messages for it are marked with `(new)`.

Figure 7 shows the typical message exchange when performing an active open: the kernel emits an `as_connect` message, which is answered after a while with an `as_okay` from the demon. After that, the connection is already established and can be used for data transmission. At the end, the connection is closed by exchanging the usual pair of `as_close` messages.

The caller in figure 8 is less lucky that the one in the previous example: the connection is rejected with `as_error`.

Figure 9 shows a simple case of closing a connection while an operation is still in progress (e.g. in a non-blocking `connect`).

Figure 10 illustrates the only case where an `as_close` is not answered with another `as_close` message: when an `as_error` was sent before. (In principle, the demon could also send an `as_error` after receiving an `as_close`.) Note that the demon has already de-registered the socket when sending the `as_error`, so the `as_close` message can't be attributed to any socket and is silently ignored.

Figure 11 illustrates creation of a listening socket, acceptance of an incoming calls, and finally closing of both sockets. First, the usual `bind` and `listen` operations are performed. After a while, the demon indicates arrival of a new connection with `as_indicate`. (Note that several `as_indicate` messages can be received in sequence, even before the first one has been accepted or rejected by the kernel.) The kernel accepts the connection with `as_accept`, which is confirmed by the signaling demon with `as_okay`. From now on, both sockets can be used in parallel.

Later on, the listening socket is closed. Note that the other socket is not affected by this and can be used until it is explicitly closed (in this case by the kernel).

Finally, figure 12 shows the case where the kernel refuses an incoming connection by sending `as_reject` (which is, of course, not acknowledged by the signaling demon). Closing of the listening socket is omitted in this diagram.

C Formal specification

Programs that describe the internal signaling protocol in a more formal way are contained in the files `caller.p` and `called.p` that are distributed with the source of this document.[9] A description of the Promela language can be found in [6, 7].

Descriptive comments are contained directly in the program source. Note that the programs have been written in a way such that they can also be used for validation, i.e. the signaling demon of the called party will send only a limited number of indications to the respective kernel side.

Because many flows are very similar for the calling and for the called side, the individual processes would be merged for an actual implementation, probably after transforming them into the corresponding state machines.

[9]See `http://lrcwww.epfl.ch/linux-atm/` for pointers to the complete package.

D Future changes

The interface described in this document will continue to evolve to support advanced signaling procedures and also to improve its semantical consistency. The following changes are being considered or are already planned at the time of this writing:

- support for multicast (point-to-multipoint) signaling as specified in UNI 4.0 [8], including the procedures that extend the capabilities beyond Q.2971 [9]
- support for modification of traffic parameters as specified in Q.2963
- possibly integration of the `aal` field into `struct atm_qos`

In order to use similar message flows for incoming connections and for multicast signaling (both share the property that sockets are temporarily "owned" by a "master" socket), the procedures for clearing of pending indications (i.e. indications for which the kernel hasn't sent an `as_accept` or an `as_close` yet) when closing a listening socket will probably be changed as follows:

- the kernel destroys pending indications after receiving the close confirmation, without further interaction with the signaling demon
- the signaling demon initiates clearing towards the network for all pending indications after acknowledging closing of the listening socket, without any further interaction with the kernel

Implementation of multicast signaling is for further study.

References

[1] Almesberger, Werner. *ATM on Linux*, `ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_on_linux.ps.gz`, EPFL, March 1996.

[2] ITU-T Recommendation Q.2931. *Broadband Integrated Services Digital Network (B-ISDN) – Digital subscriber signalling system no. 2 (DSS 2) – User-network interface (UNI) – Layer 3 specification for basic call/connection control*, ITU, February 1995.

[3] Almesberger, Werner. *Linux ATM API*, `ftp://lrcftp.epfl.ch/pub/linux/atm/api/`, EPFL, July 1996.

[4] Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application REQuested IP over ATM (AREQUIPA)* (work in progress), Internet Draft `draft-almesberger-arequipa-01.txt`, June 1996.

[5] ITU-T Recommendation Z.100. *CCITT Specification and description language (SDL)*, ITU, March 1993.

[6] Holzmann, Gerard J. *Design and validation of computer protocols*, Prentice Hall, 1991.

[7] Holzmann, Gerard J. *Basic Spin Manual*, `ftp://netlib.att.com/netlib/spin/spin2.Doc.tar.Z`

[8] The ATM Forum, Technical Committee. *ATM User-Network Interface (UNI) Signalling Specification, Version 4.0*, `ftp://ftp.atmforum.com/pub/specs/af-sig-0061.000.ps`, The ATM Forum, July 1996.

[9] ITU-T Recommendation Q.2971. *B-ISDN – DSS 2 – User-network interface layer 3 specification for point-to-multipoint call/connection control*, ITU, October 1995.

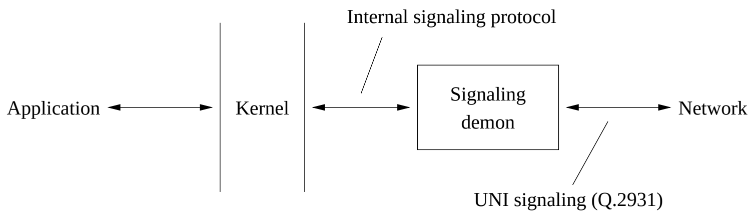


Figure 1: Linux ATM signaling concept.

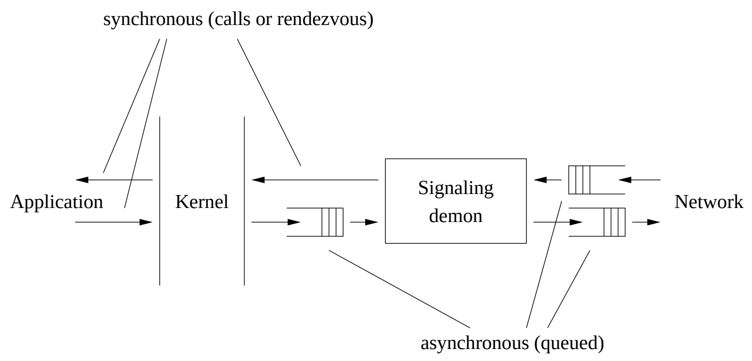


Figure 2: Linux ATM signaling concept.

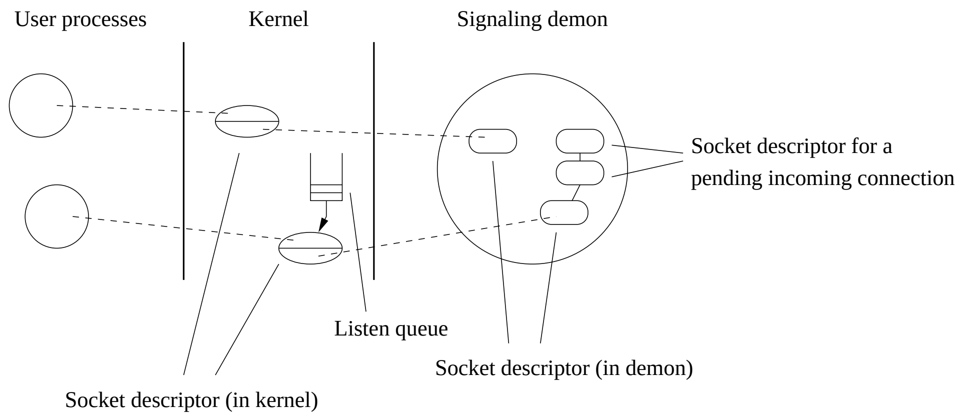
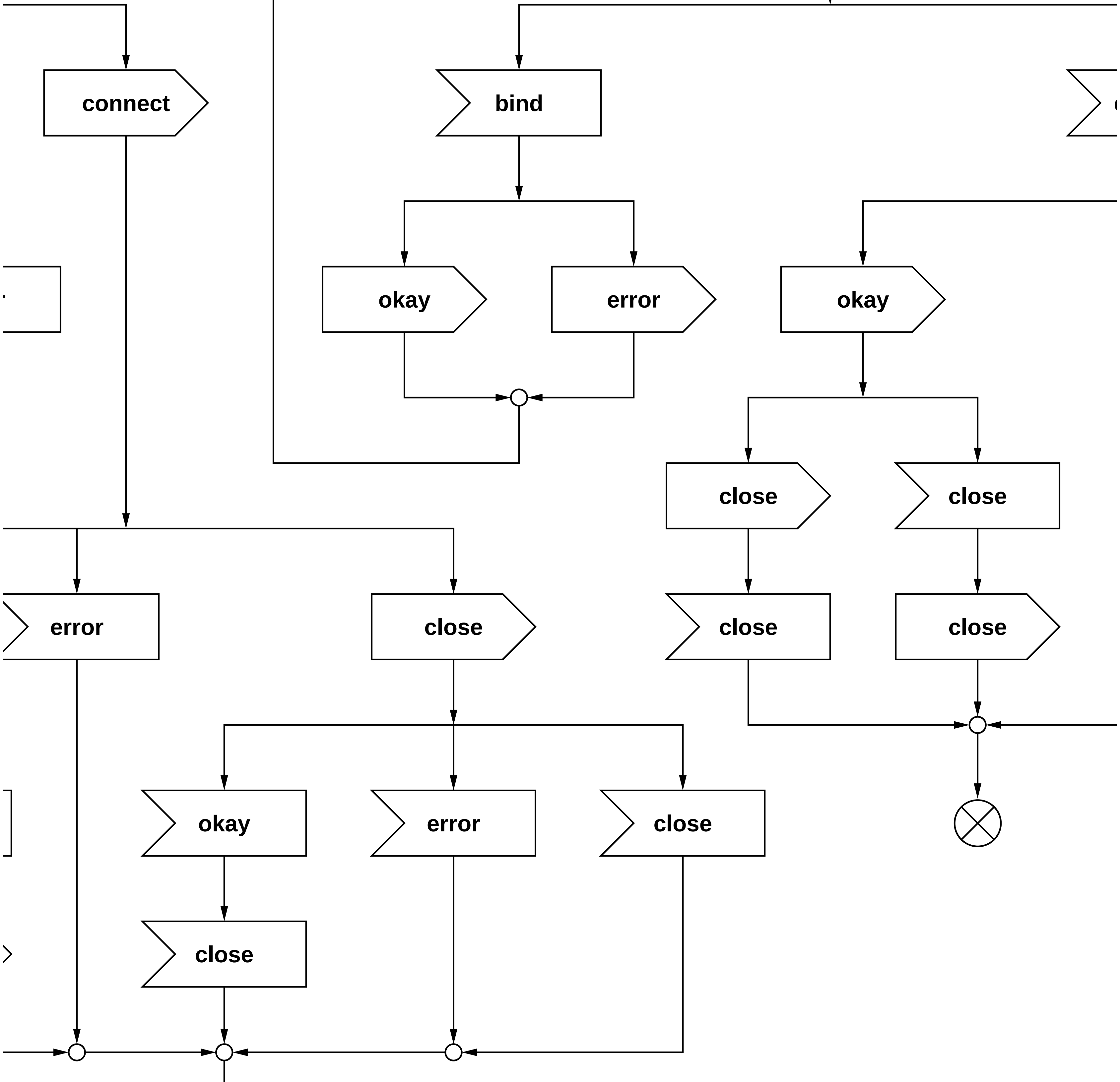
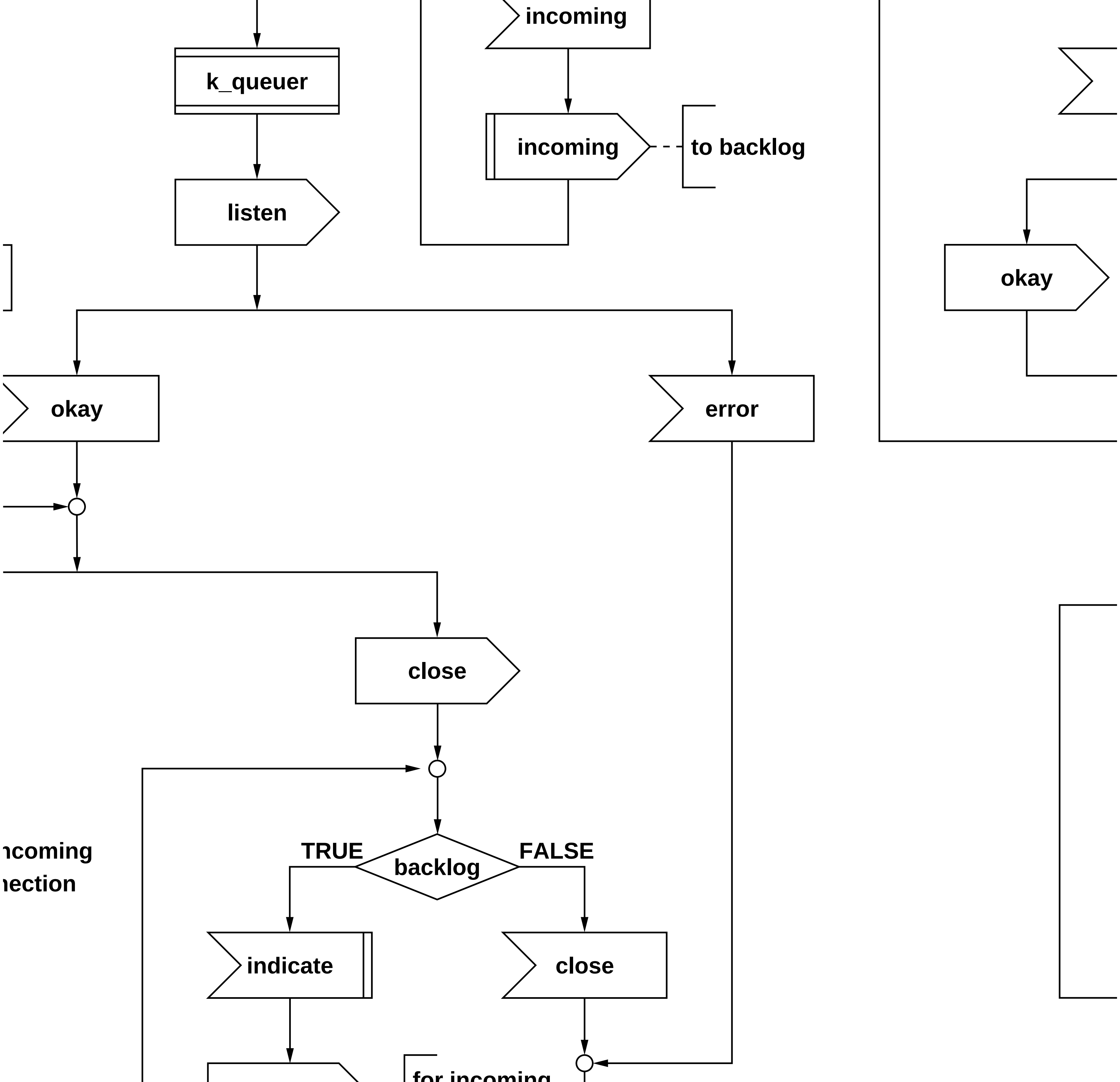
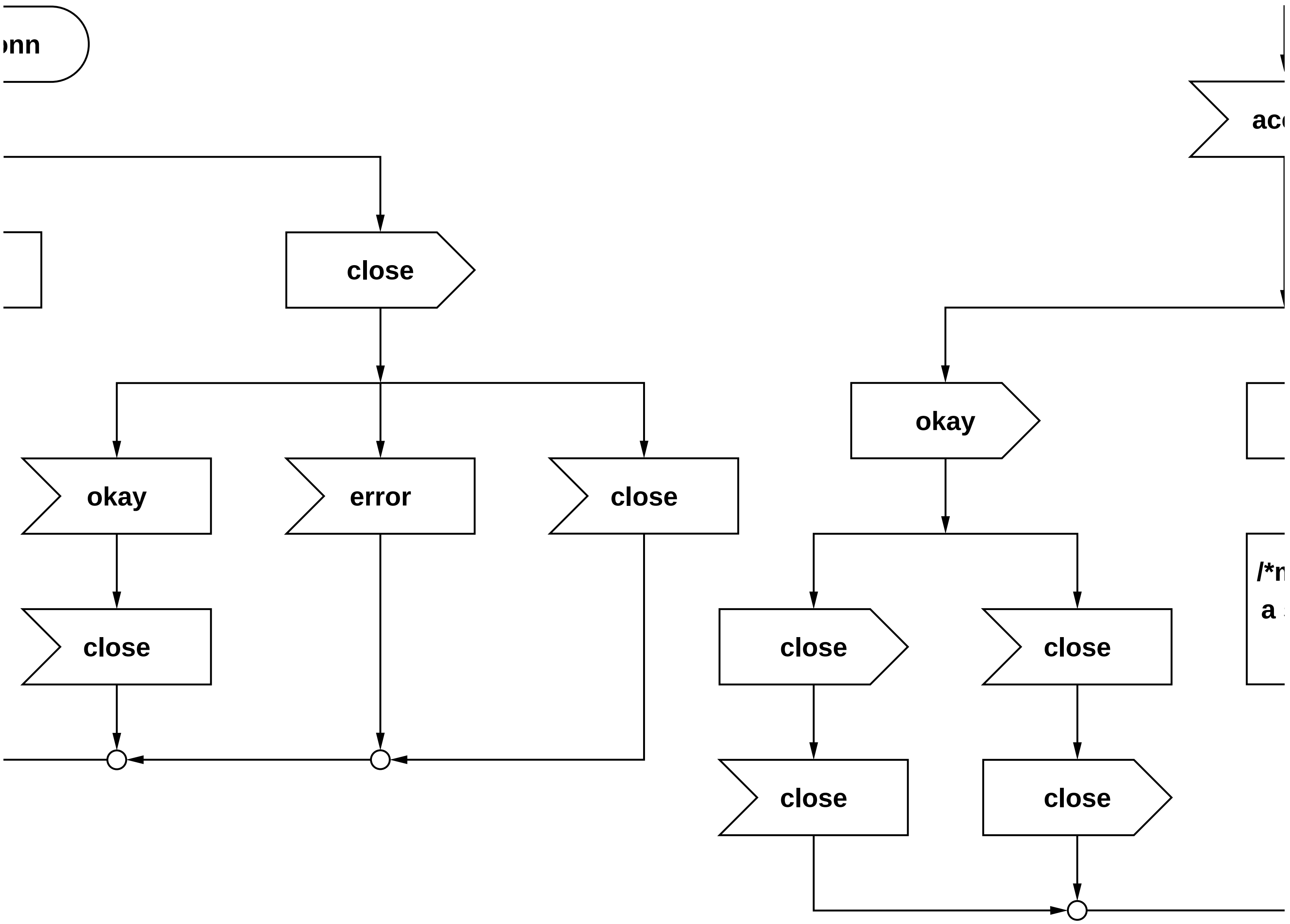

User processes
Kernel
Signaling demon
Socket descriptor for a
pending incoming connection
Listen queue
Socket descriptor (in demon)
Socket descriptor (in kernel)

Figure 3: Socket descriptors in the kernel and in the signaling demon.


connect
bind
okay
error
okay
close
close
error
close
close
close
okay
error
close
close



k_queuer
listen
incoming
incoming
to backlog
okay
okay
error
close
backlog
TRUE
FALSE
indicate
close
ncoming
nection
for incoming



onn
ac
close
okay
okay
error
close
close
close
close
close
close
/*n
a

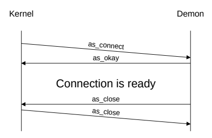


Figure 7: Normal, successful connection setup, followed by closing by the remote party.

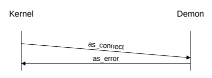


Figure 8: Unsuccessful connection setup attempt.

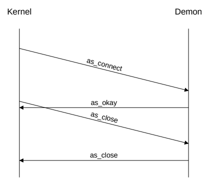


Figure 9: Crossing of `as_close` with `as_okay`.

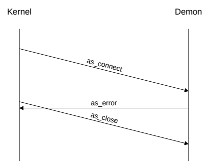


Figure 10: Crossing of `as_close` with `as_error` (which is perceived as a “stray close” by the signaling demon).

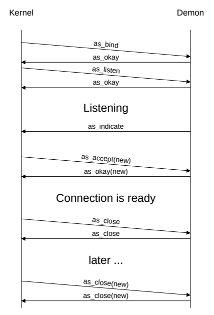


Figure 11: Creation of a listening socket and acceptance of an incoming call, followed by closing by the local party.

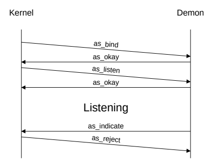


Figure 12: Creation of a listening socket and rejection of an incoming call.